Amiga Mail

Volume II

The Commodore Amiga Technical Newsletter

March/April 1991

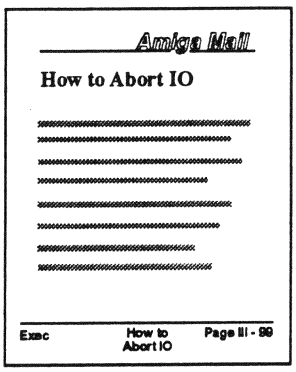
Copyright 1991 Commodore-Amiga, Inc. All rights reserved. Some of the materials used herein have been reproduced with the permission of the authors indicated. Use or reproduction of such materials shall be in accordance with the authors' instructions.

Amiga Mail

Volume II

***Amiga Mail** is published six times annually by the Commodore Applications and Technical Support Group (CATS). Its purpose is to support third-party developers of Amiga software and hardware worldwide by providing technical information and programming guidelines. Each issue features a collection of news and technical articles of special interest to Amiga product developers. The technical articles in each issue can be saved for future use in the Amiga Mail Tech Notes binder, a reference notebook containing all the articles from past issues of Amiga Mail.*

For filing technical articles in the Amiga Mail Tech Notes binder, just use the chapter and page numbers listed in the footer of each article page. The Tech Notes chapter is shown as a Roman numeral followed by the page number within that chapter.



For instance, the article at right would be filed in chapter III of the Amiga Tech Notes binder, "Exec". The article starts on page 99 of that chapter in the binder.

For more information on the Amiga Mail Tech Notes binder and the Commodore Developer Support Program, contact your local support manager or if you are in the U.S., contact CATS at (215) 431-9180.

Warning

The information contained herein is subject to change without notice. Commodore specifically does not make any endorsement or representation with respect to the use, results, or performance of the information (including without limitation its capabilities, appropriateness, reliability, currentness or availability).

Disclaimer

This information is provided "as is" without warranty of any kind, either express or implied. The entire risk as to the use of this information is assumed by the user. In no event will Commodore or its affiliated companies be liable for any damages, direct, indirect, incidental, special or consequential, resulting from any defect in the information, even if advised of the possibility of such damages.

Amiga Mail News

Double DevCon Needs Session Ideas

The next North American Developers Conference will be held September 4-7, 1991, at the Denver Marriott City Center in Denver, Colorado. That DevCon will be followed by the European DevCon, which will be held September 9-12 at the Grand Hotel Brun in Milan, Italy. In an effort to make these DevCons even better than ever, CATS is asking you, the developer, to suggest topics you would like to see discussed at the next DevCon. If you have a suggestion, you can let CATS know by mailing it to:

CATS - DevCon Suggestions
Commodore Business Machines, Inc.
1200 Wilson Drive
West Chester, PA 19380
USA

or contact Paul Higginbottom via
phone: (215) 431-9228
e-mail: higgin@cbmvax.commodore.com
or ...[uunet! rutgers]!cbmvax!higgin

Get your suggestions in quickly; work on the DevCon agenda has already started.

Toaster "Big Hit" at Macintosh Trade Show

NewTek was recently showing the Video Toaster in what some may think an unlikely place -- MacWorld. As usual, however, their booth was crowded throughout the show. They've also announced plans to exhibit at PC Expo this year.

According to the company's marketing director, Mark Randall, "The people attending these shows are looking for a product with the technology of the Video Toaster. It's not their fault they don't know they're at the wrong show; so we'll meet them there." NewTek also announced plans to continue exhibiting at COMDEX.

Reportedly, some exhibitors were miffed that the company was permitted to exhibit, but according to NewTek, Mac users responded positively to the appearance. The Toaster, which is currently shipping, has a suggested retail price of $1,595. Its output is acceptable for television broadcast applications and rivals that of dedicated machines costing $10,000 -- $50,000, according to NewTek officials.

A3000 Schematics Now Available

An A3000 schematics manual is now available to registered developers. To order, send $35 U.S. (add $2.50 for orders shipped to Canada, $5.00 for overseas orders) to CATS Orders and refer to part number 314677-02.

IN THIS ISSUE

2.0 Version Strings
Carolyn Scheppner

Using SetFunction() in a Debugger
Ewout Walraven

Introduction to Boopsi
John Orr

CATS Releases TCP/IP Networking Software

CATS has released *TCP/IP Networking Software* for the Amiga. Designed to be used in conjunction with the Commodore A2065 Ethernet card, the TCP/IP software allows you to use the TCP/IP and UDP/IP protocols as well as a number of networking applications that use these protocols. The package also includes NFS client software. This version is based in part on the Berkeley System Distribution (BSD) which provides enhanced internet support. The Commodore part number is AS225.

Addison-Wesley To Publish First Amiga Style Guide

Addison-Wesley Publishing Company plans an early spring printing of the *Amiga User Interface Style Guide* -- a manual for software developers and interface designers which describes how a standard user interface for Amiga applications should look and operate. The 200+ page style guide represents the first general compilation of interface standards by Commodore.

"Release 2 of the operating system gave the Amiga a more professional interface," said Jeff Scherb. "By introducing standards at the same time as the new operating system, we can give the Amiga a more friendly, intuitive and predictable interface. Those attributes will be crucial to our platform as computers continue to move more into the mainstream of everyday life."

The new book details the Amiga's three built-in interfaces -- the graphic user interface, the Shell and ARexx -- with definitions, descriptions and illustrations. Many of the graphic elements depicted in the book, such as gadgets and windows, are built into Release 2 of the operating system, but the detailed illustrations can also act as guides for designers working with earlier versions of the operating system. Only the behavioral guidelines are presented in the book, so, for the most part, it can be used by any interface designer regardless of his or her technical background. "The style guide will become an indispensible tool for Amiga application designers," Scherb said.

In its introduction, the book makes the point that the standards apply mainly to the controls and functions of a program and don't inhibit creativity within the application itself. "The reasons for those standards go beyond an innate need for order originating from some corporate boardroom," the manual says. "The reasons set forth here have been set forth honestly in an attempt to provide more power to the user."

North American Commercial Developers Can Put It On Plastic

North American developers registered in CATS commercial developer support program can now use VISA and MasterCard to buy hardware, software and technical reference material from CATS. The new service enables commercial developers to fax their orders to CATS and receive the merchandise much more quickly. Generally, orders can be processed the same day they are received and ship the following business day.

"The ones who have used it say they love it," said CATS' Emma Jean Mungin. "They just call up, give their order and their credit card number and that's it. They don't have to worry about anything else." The new service is only available to commercial developers.

Amiga Mail
Volume II
March/April 1991

2.0 Version Strings

By Carolyn Scheppner

Unlike the 1.3 *version* command, the 2.0 *version* command has the ability to search Amiga files for a version string. If you try the 2.0 *version* command on any of the 2.0 Workbench commands, you will find that almost all Workbench commands now contain these special version strings. For example, running *version* on the current version of *SYS:Utilities/More* will output *More 37.2*.

This embedded version string provides a simple way for a user to determine the specific version of a command. This is extremely useful for bug reports and phone support. You may enter these strings in your code yourself and update them by hand when required, or you may automate updates by using the *bumprev* tool (provided on a variety of DevCon disk sets and also in the Preliminary Software Toolkit II).

Hand-Coded Version Strings

The hand-coded method can be used in text files and is often quite suitable for simple programs with a single code module. If you code the version strings by hand, they should be formatted like the examples below. The example hand-coded strings are for a program named *myapp*, version 37.1, date 20-Mar-91 (20.3.91):

In C:

```
UBYTE versiontag[] = "\0$VER: appname 37.1 (20.3.91)";
```

In assembler:

```
versiontag      dc.b 0,'$VER: myapp 37.1 (20.3.91)',0
```

In a text file:

```
$VER: myapp.doc 37.1 (20.3.91)
```

Note that the NULL ("\0" or 0,) at the beginning of the versiontag string is not necessary but can be useful if you choose to #define the string and wish to give a version number to a C program with no data segment. With the initial NULL, you can concatenate a #defined versiontag string onto an arbitrary immediate string used in your code to get the versiontag into your code segment.

Automating Version Numbering with *Bumprev*

The *bumprev* tool and the include files it creates are what we use internally to give version numbers to system ROM modules, disk-based devices and libraries, and 2.0 Workbench and Extras commands. *Bumprev* creates or updates three files -- a *name_rev.rev* file which contains the current revision number, and the C and assembler include files called *name_rev.h* and *name_rev.i*. These include files contain #defines (.h) or macros (.i) to define the name, version, revision, and date of your program in a variety of string and numeric formats.

By using the appropriate include file in one or more of your code modules, you can use these #defines (or macros) in place of hardcoded version and revision information. This way, whenever you "bumprev" your revision files and recompile (or reassemble) your program, all version, revision, and date references in your program will be automatically updated. You can even include a *bumprev* call in your makefile for automatic revision bumping on every make (although this can update the version number more often than is really necessary).

The usage of bumprev is: `bumprev <version> <name_rev>`
For example: `bumprev 37 myapp_rev`

The first time you use the above example *bumprev* call, it creates a *myapp_rev.rev* file containing "1", and *myapp_rev.h* and *.i* files containing a variety of version and revision #defines (or macros) for version 37.1. The next time you use the same *bumprev* command it updates the files so that all #defines (or macros) are for version 37.2.

Bumprev does have some caveats. If you accidently type "bumprev 37 *myapp*" (instead of *myapp_rev*), *bumprev* will gladly overwrite any *myapp.h* or *myapp.i* file you happen to have rather than complain or automatically insert *_rev* into the output file names. Also, to make a major version switch (for example from 36 to 37), you must first delete the *myapp_rev.rev* file to make *bumprev* start the revisions over again at 1. Note that the 2.0 convention is for a major version number of 37 (to match the OS major version).

Here are example *_rev.h* and *_rev.i* files as generated by *bumprev*, and fragments of C and assembler code which include and reference these files.

Example *myapp_rev.h* generated by *bumprev*:

```
#define     VERSION     37
#define     REVISION    1
#define     DATE        "20.3.91"
#define     VERS        "myapp 37.1"
#define     VSTRING     "myapp 37.1 (20.3.91)\n\r"
#define     VERSTAG     "\0$VER: myapp 37.1 (20.3.91)"
```

Code example which includes *myapp_rev.h*:

```
/* myapp.c */
#include <exec/types.h>
#include <dos/dos.h>

/* stdlib.h and stdio.h contain prototypes for exit and printf.
 * Amiga.lib IO users could instead use <clib/alib_protos.h>
 *  and  <clib/alib_stdio_protos.h>
 */
#include <stdlib.h>
#include <stdio.h>

#include "myapp_rev.h"

/* NOTE: we reference VERSTAG version string for C:VERSION to find */
UBYTE versiontag[] = VERSTAG;

/* NOTE: we concatenate program name and version (VERS) with our copyright */
UBYTE Copyright[] = VERS " Copyright (c) 1991 CATS, Inc.  All Rights Reserved";

void main(int argc,char **argv)
    {
    /* Print our Copyright string.
     * Copyright string includes our myapp _rev.h version and date
     */
    printf("%s\n",Copyright);
    exit(RETURN_OK);
    }
```

Example *mylib_rev.i* generated by *bumprev*:

```
VERSION     EQU  37
REVISION    EQU   1
DATE        MACRO
    dc.b    '20.3.91'
    ENDM
VERS        MACRO
    dc.b    'mylib 37.1'
    ENDM
VSTRING     MACRO
    dc.b    'mylib 37.1 (20.3.91)',13,10,0
    ENDM
VERSTAG     MACRO
    dc.b    0,'$VER: mylib 37.1 (20.3.91)',0
    ENDM
```

Code example which includes *mylib_rev.i*:

```
* This is an example of an initial library code module
* Mylib_rev.i is generated with bumprev

      nolist
        include "exec/types.i"
        include "exec/initializers.i"
        include "exec/libraries.i"
        include "exec/resident.i"

        include "mylib.i"
        include "mylib_rev.i"       ; Bumprev revision include file
      list

          ; external
        xref    InitLib             ; init function
        xref    FuncTable           ; function table
        xref    EndSkip             ; End of code segment


; code at start of file in case anyone tries to execute the library as a program

        entry   FalseStart
FalseStart
        moveq   #-1,d0
        rts


ResidentNode
        dc.w    RTC_MATCHWORD       ; RT_MATCHWORD
        dc.l    ResidentNode        ; RT_MATCHTAG
        dc.l    EndSkip             ; RT_ENDSKIP
        dc.b    RTF_AUTOINIT        ; RT_FLAGS
        dc.b    VERSION             ; RT_VERSION        ;From mylib_rev.i
        dc.b    NT_LIBRARY          ; RT_TYPE
        dc.b    0                   ; RT_PRI
        dc.l    LibName             ; RT_NAME
        dc.l    IDString            ; RT_IDString       ;Contains VSTRING
        dc.l    InitTable           ; RT_SIZE           ;  from mylib_rev.i


LibName:        DC.B    'mylib.library',0
IDString:       VSTRING                                 ;From mylib_rev.i
        CNOP    0,2

InitTable
        dc.l    XMyLibBase_Size
        dc.l    FuncTable
        dc.l    DataTable
        dc.l    InitLib

DataTable
            ; standard library stuff
        INITBYTE    LN_TYPE,NT_LIBRARY
        INITLONG    LN_NAME,LibName
        INITBYTE    LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
        INITWORD    LIB_VERSION,VERSION                 ;From mylib_rev.i
        INITWORD    LIB_REVISION,REVISION               ;From mylib_rev.i
        INITLONG    LIB_IDSTRING,IDString               ;Contains VSTRING
                                                        ;  from mylib_rev.i
            ; library specific stuff

            ; end of init list
        dc.l        0

        end
```

Amiga Mail
Volume II
March/April 1991

Using SetFunction() in a Debugger

By Ewout Walraven

The Amiga OS consists of a set of libraries (and devices), which reside in ROM or on disk. These libraries provide a set of routines which are shared by the Amiga tasks (hence the name *shared* library).

The way in which an Amiga library is organized allows a programmer to change where the system looks for a library routine. Exec provides a function to do this: SetFunction(). The SetFunction() routine redirects a library function call to an application-supplied function (Although this article doesn't address it, SetFunction() can also be used on Exec devices). The *SetPatch* utility uses SetFunction(). *SetPatch* is a program which replaces some OS routines with improved ones, primarily to fix bugs in ROM libraries.

Normally, programs should not attempt to "improve" library functions. Because most programmers do not know exactly what system library functions do internally, OS patches can do more harm than good. However, a useful place to use SetFunction() is in a debugger. Using SetFunction(), a debugger can reroute library calls to a debugging function. The debugging function can inspect the arguments to a library function call before calling the original library routine (if everything is OK). Such a debugging function doesn't do any OS patching, it merely inspects.

SetFunction() is also useful for testing an application under conditions it does not encounter normally. For example, a debugging program can force a program's memory allocations to fail or prevent a program's window from opening. This allows a programmer to find bugs that only arise under special circumstances. Some programs that use SetFunction() for debugging purposes are *IO_Torture*, *Memoration* and *MungWall*. A real watchdog is *Wedge*, which, as its name implies, allows you to install a wedge for practically every function of a standard library and inform you about the register values passed to the function. These types of debugging tools helped debug release 2.0 of the OS and found bugs and 1.3 dependencies in commercial applications.

Although useful, SetFunction()ing library routines poses several problems. First of all, the wedge routine will have to be re-entrant, like all Exec library functions. Secondly, there is always a problem with removing the wedge. If another task has SetFunction()ed the same library routine as the debugger (a very real possibility), it is not normally possible to remove the first wedge, since the other task depends on the presence of your task's code. This would force your task to hang around, waiting for

the other task(s) to remove their wedges. You also need to know when it is safe to unload your debugging code. Removing it while another task is executing it will quickly lead to a hopelessly crashed system.

For those of you who might be thinking about writing down the ROM addresses returned by SetFunction() and using them in some other programs: *Forget It.* The address returned by SetFunction() is only good on the current system at the current time. Blindly jumping into ROM *will* cause your programs to break.

Exec Library Structure

When a library is opened for the first time, a library node structure, a jump table, and a data area are created in RAM.

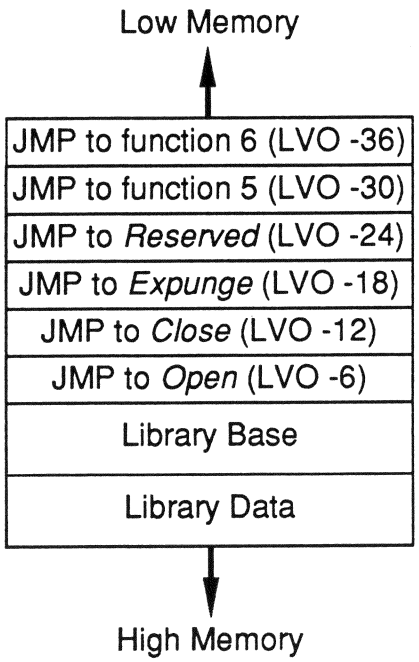

Low Memory
JMP to function 6 (LVO -36)
JMP to function 5 (LVO -30)
JMP to Reserved (LVO -24)
JMP to Expunge (LVO -18)
JMP to Close (LVO -12)
JMP to Open (LVO -6)
Library Base
Library Data
High Memory

Fig. 1
An Exec Library Base in RAM

The library node structure address is the base address of the library. OpenLibrary() returns this base address. The library's jump table, which directly precedes the library node in RAM, consists of six byte long entries containing a jump instruction (JMP) to a corresponding library function. The jump

table is initialized when Exec opens the library. Each function's entry in the jump table (also known as a vector) is always a constant (negative) offset from the library base. These fixed negative offsets are known as Library Vector Offsets (LVO). Note that the first four function vectors are reserved for use by Exec. They point to standard library functions for opening, closing, and expunging the library, plus there is space reserved for a fourth function vector. The base address of a library is determined dynamically when the library is loaded into RAM. See the Exec introduction chapter in the *ROM Kernel Manual: Libraries and Devices* for more information on libraries.

The SetFunction() routine replaces an LVO address with a new address which points to the wedge routine. SetFunction() returns the old vector address, which the wedge routine can use to call the original library function from within the wedge. Note that if another task SetFunction()s the same library function, SetFunction() returns the address of *your* debugging routine (to the second task) as the old vector. At that point your task can no longer exit since that would mean that that other task has an invalid pointer to your function and will most likely crash the system when it tries to use your function.

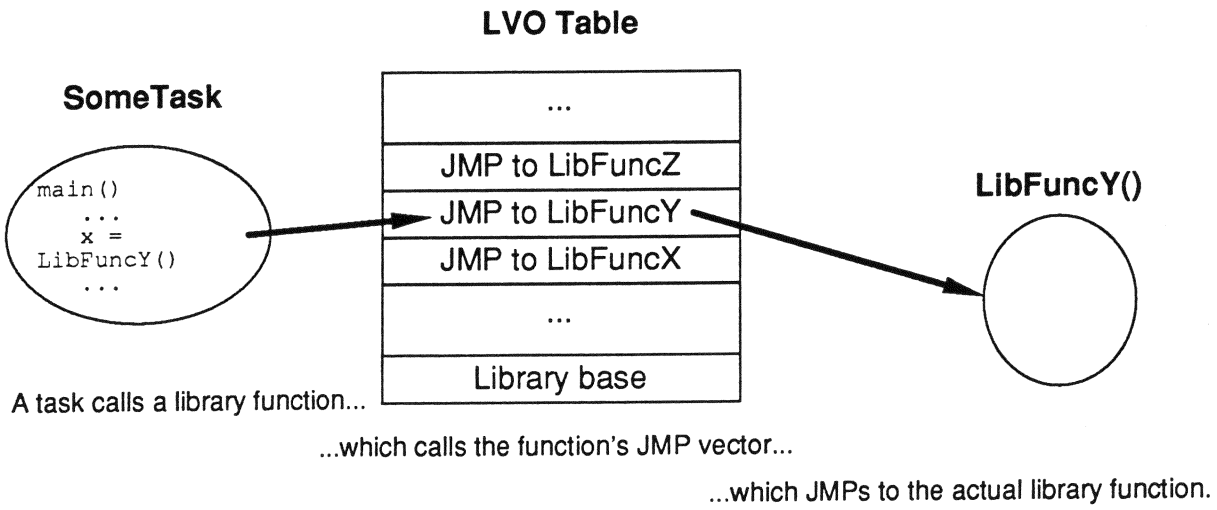


Fig. 2
Calling a Library Function

There is a way around this problem. Instead of SetFunction()ing a library function with the address of your wedge code, build your own jump table and use the addresses of its entries as arguments to SetFunction() calls. This method allows you to unload your code when you want to because if another task SetFunction()s your routine, that task will get a pointer to a jump table entry, not your routine.

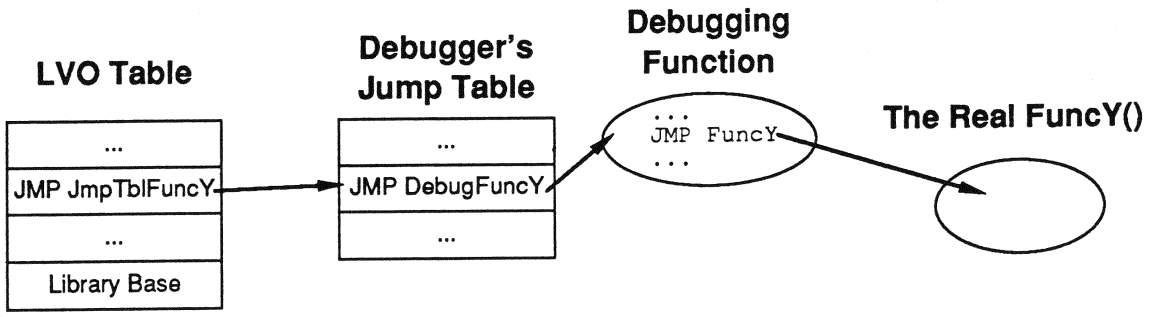


Fig. 3
Using SetFunction() with a Jump Table

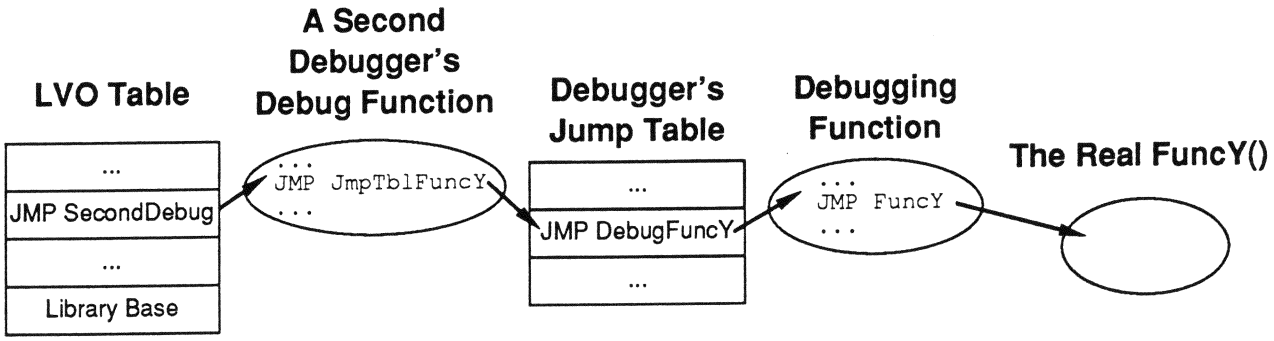


Fig. 4
Another Debugger SetFunction()s an Entry in the Jump Table

Now when you want to exit, all you need to do is replace the entries in your jump table which point to your functions with the original function vectors which were returned by SetFunction(). By not freeing the memory allocated for the jump table your task can exit any time, regardless of other tasks which SetFunction()ed the same library routines. The other task(s) will never know what happened.

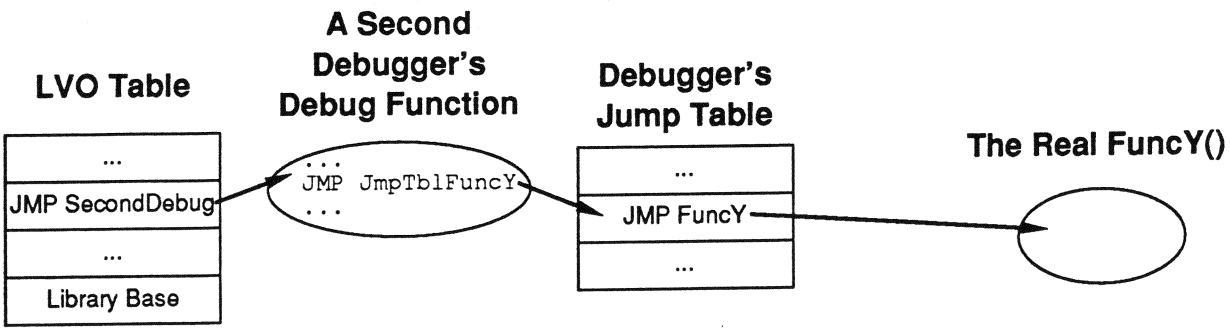


Fig. 5
Original Debugger Removes its Debugging Routine

The next time the debugger is executed, it looks for the jump table it left behind and replaces the entries in it with pointers to its own functions. Incidentally, this is an easy way to provide a mechanism to determine if your debugging program has already been installed.

Caveats

There are some things to keep in mind when using SetFunction(). The scheme described above can force a second task to hang around forever if it SetFunction()ed a routine before you, since your debugger will not normally release its handle on the second task's function. Whenever possible, install jump table based debuggers before any other SetFunction()ing program (but after *SetPatch* of course).

Some debuggers interpret the return address of the caller. When a debugger jumps (JMP) to (what it thinks is) the original function, there will be no problem. However, if a debugger performs a JSR to a second debugging function which interprets the return address, the second debugging function will receive the first debugging function's return address (the one that performed the JSR) rather than the return address of the application that called the library function in the first place. This can confuse the second debugger. Two good examples of this are *Scratcher* and *MungWall*, which both SetFunction() FreeMem(). *MungWall* looks at the return address of the caller. Since *Scratcher* calls the old FreeMem() function with a JSR instruction, it would mislead *MungWall* if run after it. Preferably, debuggers that interpret the return address should be started *after* other debuggers.

Although it is not common, some library functions call other library functions and depend on certain scratch registers to contain valid values. SetFunction()ing one of these functions is likely to change the values in these scratch registers, leading to problems. Because these dependencies are not always documented, you might innocently run into one. *Scratcher* is an excellent tool for finding such dependencies.

In the past, some system functions did not have a JMP vector in their entry in the LVO table. Instead, the actual function was in the jump table. SetFunction() will not work on such a function.

Any debugging routine should be careful not to call the function it has patched with SetFunction(), either directly or indirectly. Doing so will likely cause a stack overflow and crash the machine. This may seem a bit obvious until you consider how easy it is to indirectly call a system routine. Many system functions are not atomic. They have to call lower level system functions. If you call a higher level system functions in the debugging code and you have SetFunction()ed one of the routines the high level function uses, the machine will probably crash from a stack overflow.

Using SetFunction() on disk-based libraries and devices requires a little extra care. Unlike a ROM library, libraries (and devices) loaded into RAM can be expunged when memory gets low. To prevent the system from expunging a library (or device) you have SetFunction()ed, either keep the library (or device) opened, or use SetFunction() to patch the library's expunge function.

Remember that when you install a wedge, another program can call it almost instantly. Because of this, the wedge should be completely set up before you install it.

Note that *dos.library* is now a standard library and can be SetFunction()ed as of V36. Before V36 you would have to Forbid(), get the six original bytes of the entry in the function vector table, install the new vector, perform a SumLibrary() and then Permit().

If it is necessary to put debugging code into a Forbid() or Disable() state, keep it in that state for as little time as possible to limit system slowdown. Remember that you cannot Disable() for more than 250 microseconds. Be sure to read the Disable()/Enable() Autodocs before using them.

An Example Debugger

The usage of SetFunction() is shown by the example debugging program at the end of this article, *ISpy*. *ISpy* uses a semaphore to gain access to its jump table. This jump table contains pointers to the wedge routines. When executed, each wedge routine puts a shared lock on the semaphore to indicate that the code is being executed. To get an idea of who is calling the debugger entries, *ISpy* uses a little assembler stub to load a4 with the address of the stack of the caller and calls a C routine where the actual (simple) argument checking is done. When *ISpy* is signalled to exit, it tries to get an exclusive lock on the semaphore in a Forbid()en state. If this succeeds, it can safely assume its code is not being executed at the moment and can therefore place the original function vectors in the jump table and exit, leaving the semaphore behind. This semaphore is also used to check whether *ISpy* is already installed. If so, the new instance will exit immediately. Because of the use of shared semaphore locks, this program will only run with V37. By using a global counter (which is incremented each time a function is entered and decremented when it is exited) *ISpy* can be adapted to V33. Because of the way *ISpy* is set up, it is very easy to add argument checking front ends for functions, and have multiple versions of *ISpy* for different libraries.

Memoration and *Scratcher* by Bill Hawes. *IO_Torture* by Bryce Nesbitt. *Wedge* and *DevMon* by Carolyn Scheppner. *MungWall* by Ewout Walraven (inspired by *Memwall* by Randell Jesup and *MemMung* by Bryce Nesbitt).

```
;/* Execute me to compile with SASC 5.10a
lc -bl -cfist -d0 -O -v -j73 ispy.c
asm -iINCLUDE: ispy_stubs.asm
blink from lib:c.o ispy.o ispy_stubs.o LIB lib:amiga.lib lib:lcnb.lib lib:debug.lib SC
SD ND DEFINE __main=__tinymain
quit
**      ISpy. AmigaMail SetFunction() example.
**
**      Copyright (c) 1991 Commodore-Amiga, Inc.
**          All Rights Reserved
**
*/


#include <exec/types.h>
#include <exec/execbase.h>
#include <exec/memory.h>
#include <exec/semaphores.h>
#include <dos/dos.h>
#include <libraries/gadtools.h>
#include <string.h>
#include <dos.h>

#include <clib/dos_protos.h>
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

#define ASM __asm __saveds
#define REG(x) register __## x

#ifdef PARALLEL
#define zprintf dprintf
extern VOID     dprintf(STRPTR,...);
#else
#define zprintf kprintf
extern VOID     kprintf(STRPTR,...);
#endif

#ifdef DEBUG
#define D(x) x
#else
#define D(x) ;
#endif

/* Local protos */
VOID            main(VOID);
BOOL            InstallWedge(VOID);
BOOL            RemoveWedge(VOID);
struct JumpTable *GetJumpTable(UBYTE *);

/* Assembler stubs will return a pointer to the caller's stack in a4.
 * The only system function at this moment using a4 is the workbench.library
 * AddAppIconA() function.
 */
#define ACALLER (0)             /* StackPtr[0] = ACaller, StackPtr[1] = saved A6 in
stub */
#define CCALLER (2)             /* StackPtr[2] = CCaller */

/* The number of 'replacement' functions */
#define NUMBEROFFUNCTIONS (4)

/* prototypes for the functions to be SetFunction()'ed. */

/* intuition.library */
struct Screen  *(*ASM oldOpenScreen) (REG(a0) struct NewScreen *,
                                      REG(a6) struct Library *);
struct Window  *(*ASM oldOpenWindowTagList) (REG(a0) struct NewWindow *,
                                             REG(a1) struct TagItem *,
                                             REG(a6) struct Library *);
struct Screen  *ASM newOpenScreen(REG(a0) struct NewScreen *,
                                  REG(a4) ULONG *,
                                  REG(a6) struct Library *);
struct Window  *ASM newOpenWindowTagList(REG(a0) struct NewWindow *,
                                         REG(a1) struct TagItem *,
                                         REG(a4) ULONG *,
                                         REG(a6) struct Library *);

/* exec.library */
VOID(*ASM oldFreeMem) (REG(a1) VOID *,
                       REG(d0) ULONG,
                       REG(a6) struct Library *);
VOID ASM        newFreeMem(REG(a1) VOID *,
                           REG(d0) ULONG,
                           REG(a4) ULONG *,
                           REG(a6) struct Library *);

/* graphics.library */
VOID(*ASM oldSetFont) (REG(a1) struct RastPort *,
                       REG(a0) struct TextFont *,
                       REG(a6) struct Library *);
VOID ASM        newSetFont(REG(a1) struct RastPort *,
                           REG(a0) struct TextFont *,
                           REG(a4) ULONG *,
                           REG(a6) struct Library *);

/* Assembler Stubs */
extern          OpenScreenStub();
extern          OpenWindowTagListStub();
extern          FreeMemStub();
extern          SetFontStub();

/* The LVO's to use from amiga.lib */
extern          LVOOpenScreen;
extern          LVOOpenWindowTagList;
extern          LVOFreeMem;
extern          LVOSetFont;

extern struct ExecBase *SysBase;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

/* Use a table and an array to make it a little more generic and easier to
 * add functions.
 */
struct LVOTable
{
    LONG            lt_LVO;
    struct Library *lt_LibBase;
    ULONG           lt_oldFunction;
    ULONG           lt_newFunction;
};

struct LVOTable LVOArray[] =
{
    {&LVOOpenScreen, (struct Library *) &IntuitionBase,
        &oldOpenScreen,
        &OpenScreenStub},
    {&LVOOpenWindowTagList, (struct Library *) &IntuitionBase,
                        &oldOpenWindowTagList, &OpenWindowTagListStub},
    {&LVOFreeMem, (struct Library *) & SysBase, &oldFreeMem, &FreeMemStub},
    {&LVOSetFont, (struct Library *) & GfxBase, &oldSetFont, &SetFontStub},
};

struct JumpTable
{
    struct SignalSemaphore jt_Semaphore;
    UWORD           pad_word;
    struct Task    *jt_Owner;
    UBYTE           jt_Function[NUMBEROFFUNCTIONS * 6];
};

/* Strings */
/* The name this JumpTable/Semaphore will get. */
static UBYTE   *JTName = "ISpy-MiscJumpTable";

static UBYTE   *VersTag = "\0$VER: ISpy 37.4 (21.3.91)";
static UBYTE   *VerTitle = "Ispy 37.4";
static UBYTE   *Copyright = "Copyright (c) 1991 Commodore-Amiga, Inc.\n";
static UBYTE   *CBreak = "CTRL-C or BREAK to exit...\n";
```

```
VOID
main(VOID)
{
    Write(Output(), (STRPTR) VerTitle, strlen((STRPTR) VerTitle));
    Write(Output(), Copyright, strlen(Copyright));

    if (SysBase->LibNode.lib_Version > 36)
    {
        if (IntuitionBase = OpenLibrary("intuition.library", 37))
        {
            if (GfxBase = OpenLibrary("graphics.library", 37))
            {
                if (InstallWedge())
                {
                    Write(Output(), CBreak, strlen(CBreak));
                    Wait(SIGBREAKF_CTRL_C);
                    RemoveWedge();
                }
                CloseLibrary(GfxBase);
            }
            else
                Write(Output(), "Couldn't open graphics.library\n", 31);
            CloseLibrary(IntuitionBase);
        }
        else
            Write(Output(), "Couldn't open intuition.library\n", 32);
    }
    else
        Write(Output(), "Requires at least Kickstart 2.04\n", 33);
}


BOOL
InstallWedge(VOID)
{
    struct JumpTable *jumptable;
    ULONG           *addressptr;
    UCOUNT           i, j;

    Forbid();

    /* Get pointer to JumpTable. Create it if necessary */
    if (jumptable = GetJumpTable(JTName))
    {
        /* Try to get exclusive lock on semaphore, in case it already existed. */
        if (AttemptSemaphore((struct SignalSemaphore *) jumptable))
        {
            /* Make sure nobody else has function addresses in the jumptable */
            if (jumptable->jt_Owner == NULL)
            {
                jumptable->jt_Owner = FindTask(0);
                /* Don't want to disable any longer than necessary */
                Disable();

                for (i = 2, j = 0; i < NUMBEROFFUNCTIONS * 6; i += 6, j++)
                {
                    /* Replace addresses in the jumptable with my own. */
                    addressptr = (ULONG *) ((UBYTE *) jumptable->jt_Function + i);
                    (*((ULONG *) LVOArray[j].lt_oldFunction)) = (ULONG) * addressptr;
                    *addressptr = (ULONG) LVOArray[j].lt_newFunction;
                    D(zprintf("setting table to Function: 0x%lx\n", *addressptr));
                }
                Enable();
            }
            else
                Write(Output(), "Already running.\n", 16);
            ReleaseSemaphore((struct SignalSemaphore *) jumptable);
        }
        else
            Write(Output(), "Can't lock table.\n", 18);
    }
    else
        Write(Output(), "Can't create jumptable\n", 23);
    Permit();
    return ((BOOL) jumptable);
}
```

```
BOOL
RemoveWedge(VOID)
{
    struct JumpTable *jumptable;
    ULONG           *addressptr;
    UCOUNT           i, j;

    Forbid();

    if (jumptable = GetJumpTable(JTName))
    {
        D(zprintf("jumptable @ 0x%lx\n", jumptable));

        /* Check if this task owns this jumptable */
        if (jumptable->jt_Owner == FindTask(0))
        {

            /* Get the semaphore exclusively.
             * Depending on what got SetFunction()'ed this could take some time.
             * Also note that shared locks are used to indicate the code is
             * being executed and that shared locks can jump ahead of queue'ed
             * exclusive locks, adding to the waittime.
             */
            ObtainSemaphore((struct SignalSemaphore *) jumptable);

            Disable();

            /* Restore old pointers in jumptable */

            for (i = 2, j = 0; i < NUMBEROFFUNCTIONS * 6; i += 6, j++)
            {
                addressptr = (ULONG *) ((UBYTE *) jumptable->jt_Function + i);
                *addressptr = (*((ULONG *) LVOArray[j].lt_oldFunction));
                D(zprintf("setting table to oldFunction: 0x%lx\n", *addressptr));
            }

            Enable();

            jumptable->jt_Owner = NULL;
            ReleaseSemaphore((struct SignalSemaphore *) jumptable);
        }
    }

    Permit();

    return (TRUE);
}

struct JumpTable *
GetJumpTable(UBYTE * name)
{
    struct JumpTable *jumptable;
    ULONG           *addressptr;
    UWORD           *jmpinstr;
    UBYTE           *jtname;
    UCOUNT           i, j;

    /* Not really necessary to forbid again, just to indicate that I don't
     * want another task to create the semaphore while I'm trying to do the
     * same. Here GetJumpTable() is only called from InstallWedge(), so it
     * will just bump the forbid count.
     */
    Forbid();

    if (!(jumptable = (struct JumpTable *) FindSemaphore(name)))
    {
        if (jumptable = AllocMem(sizeof(struct JumpTable), MEMF_PUBLIC | MEMF_CLEAR))
        {
            if (jtname = AllocMem(32, MEMF_PUBLIC | MEMF_CLEAR))
            {

                for (i = 0, j = 0; i < NUMBEROFFUNCTIONS * 6; i += 6, j++)
                {
                    jmpinstr = (UWORD *) ((UBYTE *) jumptable->jt_Function + i);
                    *jmpinstr = 0x4EF9;
```

```
                addressptr = (ULONG *) ((UBYTE *) jumptable->jt_Function + i + 2);
                *addressptr = (ULONG) SetFunction(
                    (struct Library *) (*((ULONG *) LVOArray[j].lt_LibBase)),
                    LVOArray[j].lt_LVO,
                    (VOID *) ((UBYTE *) jumptable->jt_Function + i));
            }

            jumptable->jt_Semaphore.ss_Link.ln_Pri = 0;

            strcpy(jtname, name);
            jumptable->jt_Semaphore.ss_Link.ln_Name = jtname;
            AddSemaphore((struct SignalSemaphore *) jumptable);
            /* In 1.2/1.3 AddSemaphore() didn't work properly.
            ** Under 1.2/1.3, change it to:
            ** InitSemaphore(jumptable);
            ** Forbid();
            ** Enqueue(&SysBase->SemaphoreList, jumptable);
            ** Permit();
            */
        }
        else
        {
            FreeMem(jumptable, sizeof(struct JumpTable));
            jumptable = NULL;
        }
    }
}

Permit();

/* If succeeded, you now have a jumptable which entries point to the original
 * library functions. If another task SetFunction()'ed one or more of those
 * already, that task can never go away anymore.
 */
return (jumptable);
}


/* Note: if you'd want this to work with 1.3, you wouldn't/couldn't lock the
 * semaphore, but instead would have to use a global to in- and decrement.
 * When exiting, you'd spin around the global counter, waiting for it to reach
 * zero. At that point you'd Disable(), reset the pointers in the jumptable and
 * Enable() again.
 */


struct Screen  *ASM
newOpenScreen(REG(a0) struct NewScreen * newscreen,
              REG(a4) ULONG * stackptr,
              REG(a6) struct Library * base)
{
    struct SignalSemaphore *jt;
    struct Screen  *screen = NULL;

    /* Find the semaphore to lock shared, indicating the routine is being run. */
    /* For speed reasons you may want to cache the pointer to the semaphore
    ** in a global variable */
    if (jt = FindSemaphore(JTName))
    {
        /* Lock shared in 2.0. In 1.3 you'd increment a global counter */
        ObtainSemaphoreShared(jt);
        /* Simple test for valid argument. Could check all the fields too. */
        if (newscreen != NULL)
        {
            screen = (*oldOpenScreen) (newscreen, base);
        }
        else
        {
            ULONG           ACaller = stackptr[ACALLER];
            ULONG           CCaller = stackptr[CCALLER];

            Forbid();           /* To make sure the output isn't garbled */
            zprintf("OpenScreen(NULL) by '%s' (at 0x%lx) from A:0x%lx C:0x%lx
SP:0x%lx\n",
                    SysBase->ThisTask->tc_Node.ln_Name,
                    SysBase->ThisTask,
                    ACaller,
                    CCaller,
                    stackptr);
            Permit();
        }
        /* Release shared lock. In 1.3  you'd decrement the global pointer */
        ReleaseSemaphore(jt);
    }

    return (screen);
}

struct Window  *ASM
newOpenWindowTagList(REG(a0) struct NewWindow * newwindow,
                     REG(a1) struct TagItem * tags,
                     REG(a4) ULONG * stackptr,
                     REG(a6) struct Library * base)
{
    struct SignalSemaphore *jt;
    struct Window  *window = NULL;


    if (jt = FindSemaphore(JTName))
    {
        ObtainSemaphoreShared(jt);
        if (newwindow != NULL || tags != NULL)
        {
            window = (*oldOpenWindowTagList) (newwindow, tags, base);
        }
        else
        {
            ULONG           ACaller = stackptr[ACALLER];
            ULONG           CCaller = stackptr[CCALLER];

            Forbid();
            zprintf("OpenWindowTagList(NULL,NULL) by '%s' (at 0x%lx) from A:0x%lx
C:0x%lx SP:0x%lx\n",
                    SysBase->ThisTask->tc_Node.ln_Name,
                    SysBase->ThisTask,
                    ACaller,
                    CCaller,
                    stackptr);
            Permit();
        }
        ReleaseSemaphore(jt);
    }

    return (window);
}


VOID ASM
newFreeMem(REG(a1) VOID * memptr,
           REG(d0) ULONG size,
           REG(a4) ULONG * stackptr,
           REG(a6) struct Library * base)
{
    struct SignalSemaphore *jt;

    if (jt = FindSemaphore(JTName))
    {
        ObtainSemaphoreShared(jt);
        if (memptr != NULL && size != 0L)
        {
            (*oldFreeMem) (memptr, size, base);
        }
        else
        {
            ULONG           ACaller = stackptr[ACALLER];
            ULONG           CCaller = stackptr[CCALLER];

            Forbid();
            zprintf("FreeMem(0x%lx,%ld) by '%s' (at 0x%lx) from A:0x%lx C:0x%lx
SP:0x%lx\n",
                    memptr, size,
                    SysBase->ThisTask->tc_Node.ln_Name,
                    SysBase->ThisTask,
                    ACaller,
```

```
                    CCaller,
                    stackptr);
                Permit();
            }

            ReleaseSemaphore(jt);
        }
}

VOID ASM
newSetFont(REG(a1) struct RastPort * rp,
           REG(a0) struct TextFont * font,
           REG(a4) ULONG * stackptr,
           REG(a6) struct Library * base)
{
    struct SignalSemaphore *jt;


    if (jt = FindSemaphore(JTName))
    {
        ObtainSemaphoreShared(jt);
        if (rp != NULL && font != NULL)
        {
            (*oldSetFont) (rp, font, base);
        }
        else
        {
            ULONG           ACaller = stackptr[ACALLER];
            ULONG           CCaller = stackptr[CCALLER];

            Forbid();
            zprintf("SetFont(0x%lx,0x%lx) by '%s' (at 0x%lx) from A:0x%lx C:0x%lx
SP:0x%lx\n",
                    rp, font,
                    SysBase->ThisTask->tc_Node.ln_Name,
                    SysBase->ThisTask,
                    ACaller,
                    CCaller,
                    stackptr);
            Permit();
        }
        ReleaseSemaphore(jt);
    }
}
```

```
**
**      ISpy stubs
**
**      Copyright (c) 1991 Commodore-Amiga, Inc.
**          All Rights Reserved
**
**

        INCLUDE "exec/types.i"

        SECTION CODE

        XREF    _SysBase

        XREF     _LVOForbid
        XREF     _LVOPermit

        XREF    _newOpenScreen
        XREF    _newOpenWindowTagList
        XREF    _newFreeMem
        XREF    _newSetFont

        XDEF    _OpenScreenStub
        XDEF    _OpenWindowTagListStub
        XDEF    _FreeMemStub
        XDEF    _SetFontStub


_OpenScreenStub:
        movem.l     a4,-(sp)
        lea         4(sp),a4
        jsr         _newOpenScreen
        movem.l     (sp)+,a4
        rts

_OpenWindowTagListStub:
        movem.l     a4,-(sp)
        lea         4(sp),a4
        jsr         _newOpenWindowTagList
        movem.l     (sp)+,a4
        rts

_FreeMemStub:
        movem.l     a4,-(sp)
        lea         4(sp),a4
        jsr         _newFreeMem
        movem.l     (sp)+,a4
        rts

_SetFontStub:
        movem.l     a4,-(sp)
        lea         4(sp),a4
        jsr         _newSetFont
        movem.l     (sp)+,a4
        rts

        END
```

Introduction to Boopsi

By John Orr

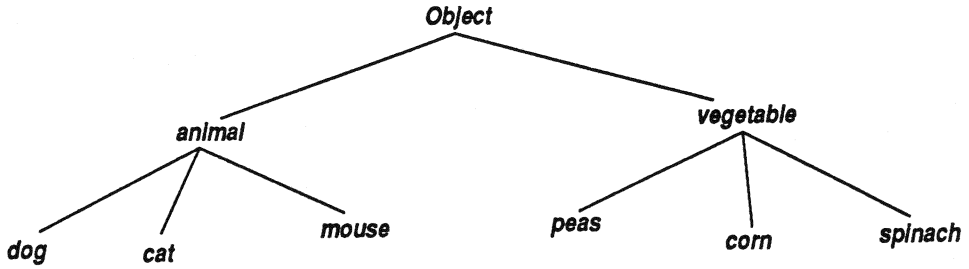
Boopsi is an acronym for Basic Object Oriented Programming System for Intuition. On its simplest level, boopsi allows the application programmer to create Intuition supported gadgets and images with minimal overhead. It allows a program to consolidate gadgets into one entity to make processing and updating easy. On a more sophisticated level, boopsi provides ways to create a wide variety of system supported, extensible custom gadgets.

Understanding boopsi requires an understanding of several of the concepts behind Object Oriented Programming (OOP). This article only briefly covers those concepts. For a more in depth explanation of those concepts, see Timothy Budd's book titled *A Little Smalltalk* (Addison-Wesley Publishing ISBN 0-201-10698-1). A port (by Bill Kinnersley) of Timothy Budd's Little Smalltalk interpreter is on Fish disk #37.

In the boopsi version of the Object Oriented Programming (OOP) model, everything is an *Object.* Each object is a distinct entity. Certain objects have similar characteristics and can be classified into different groups called *classes*. As objects, a dog, a cat, and a mouse are all distinct objects but they all have something in common; they can all be classified as animals. A specific object is an *instance* of a particular class ("Rover" is an instance of class "dog").

Classes can be subdivided into *subclasses*. A vegetable object class can have several subclasses such as peas, corn, and spinach. Inversely, the *superclass* of peas, corn, and spinach is "vegetable". In turn, both the "animal" and "vegetable" classes are subclasses. They are subclasses of a universal root category. The OOP language *Smalltalk* calls this class "Object".

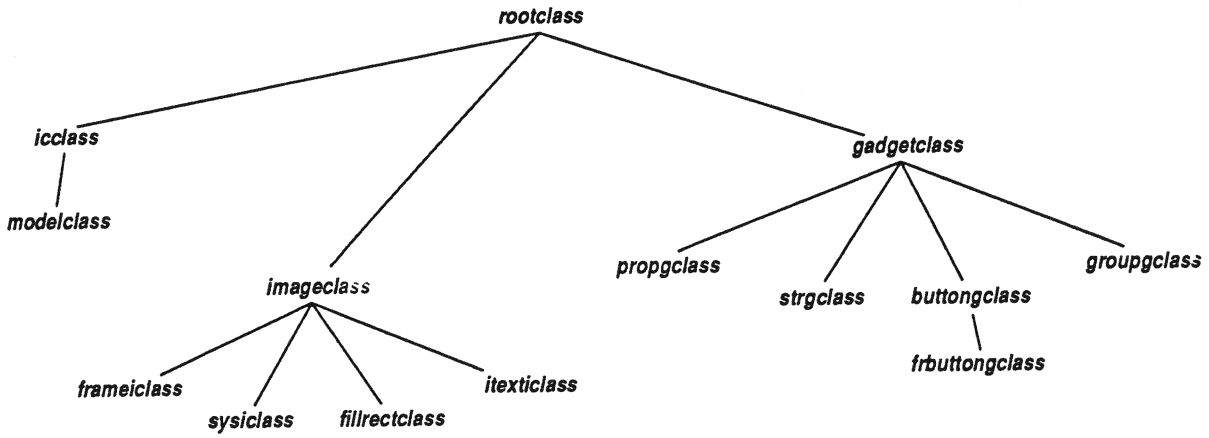
Figure 1 - Classes



In boopsi, the universal root category is called *rootclass*. Intuition supplies three subclasses of *rootclass*: *gadgetclass*, *imageclass*, and *icclass*. All boopsi gadgets in the system are an instance of *gadgetclass*. Likewise, all boopsi images are an instance of *imageclass*. The remaining subclass, *icclass*, is a concept new to Intu. that allow one boopsi object (such as a gadget) to notify another boopsi object when some specific event occurs. For example, a program can offer a user two methods of altering one integer value; one by sliding a proportional gadget, the other by typing in a value at a string gadget. Without boopsi, the program would have to explicitly update one gadget when the other was altered by the user. Using a boopsi *icclass* object, the gadgets can update each other. The gadgets update each other automatically, without the calling program's intervention.

The *rootclass*' subclasses each have their own subclasses. These subclasses are discussed later in this article.

Figure 2 - The Boopsi Classes



It is possible for an application to create its own *custom* boopsi class. When an application creates a class, it can make it either public or private. A public class has a name associated with it (for example *gadgetclass*) so arbitrary applications can access it. A private class has no name associated with it, so unless an application has a pointer to the class, it cannot access it.

Each class has a set of *methods* associated with it Each method of a class is an operation that applies to objects of that class. A example of a method for a class of integers is to add or to subtract "integer" objects. All boopsi actions are carried out via methods.

In OOP terminology, an application requests an object to perform some method by sending the object a *message* (which is not related to the Exec style message). In boopsi, the *amiga.lib* function DoMethod() accepts a method ID and some method specific parameters (older versions of *amiga.lib* do not have DoMethod(), but the function is available on the Atlanta DevCon disks in *classface.o*). DoMethod() creates a message from these parameters and sends the message to the object:

```
ULONG DoMethod(Object *myobject, ULONG MethodID, ...);
```

where myobject is a pointer to the target object, MethodID specifies which method to perform. Any remaining parameters are passed on to the method in the form of a boopsi message. Each message contains the method ID and the parameters for the method. The parameters are method specific. For example, one way to delete the *imageclass* object *obj2delete* is to call the DoMethod() function like this:

```
DoMethod(obj2delete, OM_DISPOSE);
```

The `OM_DISPOSE` method does not require any arguments, so, in this case, DoMethod() has only two arguments.

One peculiar thing about the above function call is that the `OM_DISPOSE` method is not defined for *imageclass*. It is instead defined in the superclass of *imageclass*, *rootclass*. The `OM_DISPOSE` method works because *imageclass inherits* the methods of its superclass, *rootclass*. If an object is asked to perform a method that its class does not explicitly define, it passes the request onto its superclass for processing. This superclass can in turn pass on the request to its superclass if it does not understand the method requested.

Some of the methods currently defined for Boopsi's *rootclass* are:

`OM_NEW` - Creates a new object. Normally this method is not called directly by the application programmer (a.k.a. using the DoMethod() function). Instead, there is an Intuition function called NewObject() that takes care of creating objects.

```
APTR NewObject( struct IClass *privateclass, UBYTE *publicclassID, unsigned long tag1, ...);
```

The privateclass and publicclassID parameters are used to specify the new object's class. NewObject() only pays attention to one of these. If privateclass is `NULL`, publicclassID points to a string naming the

class of the new object. If privateclass is not NULL, it contains a pointer to a private class. The remaining arguments make up a series of tag ID/data pairs. These are used to set the object's default attributes. These attributes are specific to the class in question. This function returns a handle to the new object.

Each object (or *instance* of a class) has *instance data* associated with it. The OM_NEW method takes care of allocating memory for the instance data for each class. Instance data is class specific, but all subclasses inherit instance data from their superclasses. For example, part of the instance data for a gadgetclass object is a Gadget structure. Any objects that are instances of subclasses of *gadgetclass* have a Gadget structure embedded in them. Subclasses can have their own class specific instance data in addition to the instance data inherited from the superclass.

Boopsi *gadgetclass* and *imageclass* objects are organized so that NewObject() returns a pointer to the corresponding Intuition structure embedded within them (struct Gadget and struct Image, respectively). This makes it possible to use non-boopsi Intuition functions on boopsi objects. Normally, these internal structures should be considered private, and should be accessed through the corresponding attributes, but if it is necessary, an application can look at certain fields in the embedded structure. For both gadgets and images, the Left, Top, Width, and Height fields are legal to look at. The Images NextImage field is also OK for viewing. However, this does not mean that it is OK for an application to go poking around the internals of boopsi objects. All boopsi objects are strictly private and can change without notice. Use only the functions Intuition provides for manipulating boopsi objects.

Another function, NewObjectA(), works exactly like NewObject(), but instead of accepting the tag pairs as arguments, it accepts a single pointer to a TagList. See the Intuition Autodocs for more details.

OM_DISPOSE - Deletes an object. Like OM_NEW this method is not normally called directly by the application program. Instead there is an Intuition function DisposeObject() that takes care of object disposal.

```
void DisposeObject( APTR object2delete );
```

OM_SET - Sets object specific attributes. This method is not normally called by the application program directly. Instead, there are two Intuition functions that set object attributes:

```
ULONG SetAttrs( APTR object, unsigned long tag1, ... );
ULONG SetGadgetAttrs( struct Gadget *mygadgetobject, struct Window *window,
      struct Requester *requester, unsigned long tag1, ... );
```

SetAttrs() accepts as arguments a pointer to the object in question and a series of tag pairs corresponding to the attributes to set. SetGadgetAttrs() is a special version of SetAttrs() that is required to change the attributes of *gadgetclass* objects. SetGadgetAttrs() is similar to SetAttrs(), except it has some extra parameters that a gadget needs to redraw itself in response to the attribute changes. This function can be used on non-gadgetclass objects as the gadget specific parameters are

ignored by the other object classes. If the gadget is not yet attached to a window or requester, these arguments should be set to NULL.

Both of these functions have corresponding TagList based arguments, SetAttrsA() and SetGadgetAttrs().

OM_GET - Reads an object specific attribute. The Intuition function GetAttr() provides easy access to this method:

```
ULONG GetAttr( unsigned long attrID, APTR object, ULONG *storagePtr );
```

GetAttr() fills in storagePtr with the value of the object's attribute attrID. The function returns FALSE if there is an error.

OM_ADDMEMBER - Adds a boopsi object to another object's list, if it has one. Certain object classes have an Exec list as part of their instance data. To add the object object2add to the list of the object mainobject use DoMethod() like so:

```
DoMethod(mainobject, OM_ADDMEMBER, object2add);
```

DoMethod() also has a non-varargs form called DM() (also in *amiga.lib*). DM() accepts an object pointer and a pointer to a structure specific to a method. For example, the DM() form of the above DoMethod() call would look like this:

```
DM(mainobject, addmemstruct);
```

where addmemstruct is a pointer to the following structure (defined in <intuition/classusr.h>):

```
struct opMember {
    ULONG        MethodID;      /* in this case MethodID = OM_ADDMEMBER */
    Object       *opam_Object;  /* = addmemstruct */
};
```

OM_REMMEMBER - Removes a boopsi object previously added with OM_ADDMEMBER. The parameters are the same for OM_ADDMEMBER.

OM_UPDATE - Updates attributes of an object. For gadgets, this method is very similar to the OM_SET method. It is normally used only between objects for notifying each other of attribute changes, so simple boopsi users should use the SetAttrs() and SetGadgetAttrs() functions.

OM_NOTIFY - Notifies one object when another object's attribute(s) have changed. It is normally used only between objects for modifying each others attributes, so simple boopsi users should use the SetAttrs() and SetGadgetAttrs() functions.

Attributes

All boopsi objects have attributes associated with them. These attributes reflect various properties of a specific object. For example, an image object has attributes such as IA_Left, IA_Top, IA_Width, and IA_Height, all of which correspond to fields in the Image structure. Five methods deal with an object's attributes:

`OM_NEW` sets attributes at object creation (initialization) time
`OM_SET` changes attributes after the object has been created
`OM_GET` reads an object attribute
`OM_UPDATE` updates an object attribute (for use by objects only)
`OM_NOTIFY` notifies one object of changes to another object's attributes

Not all of these methods apply to all attributes. Some attributes are not "settable" or not "gettable", for example. See the appendix at the end of this article to find out which of these methods apply to specific attributes.

Imageclass Subclasses

An *imageclass* object contains an Image structure that Intuition uses to render objects such as gadgets. Boopsi *imageclass* objects are organized so that when NewObject() returns a pointer to an *imageclass* object, it points to the actual Image structure.

Normally, an application does not create an *imageclass* object. Instead, it will use a subclass of *imageclass*. Currently, there are four subclasses: *frameiclass*, *sysiclass*, *fillrectclass*, and *itexticlass*.

frameiclass - An embossed or recessed rectangular frame, rendered in the proper DrawInfo pens, with enough intelligence to bound or center its contents.

sysiclass - The class of system images. The class includes all the images for the system gadgets, and the Gadtools check mark and button glyphs.

fillrectclass - Rectangle with frame and pattern support.

itexticlass - A specialized image class used for rendering text. Note that you have to calculate *itexticlass* object's width and height yourself.

Gadgetclass Subclasses

A *gadgetclass* object contains an Intuition Gadget structure. Like *imageclass*, applications do not normally create objects of this class, but instead create objects of subclasses of *gadgetclass*. Currently, *gadgetclass* has four subclasses:

propgclass - An easy to use, one-dimensional proportional gadget.
strgclass - A string gadget.
groupgclass - A special gadget class that creates one composite gadget out of several others.
buttongclass - A repeating boolean button gadget.

buttongclass has a subclass of its own:

frbuttongclass - A button that outlines its label with a frame.

The example *boopsi1.c* (at the end of this article) uses *sysiclass* images and several boopsi gadgets. The example takes care of processing and updating the gadgets.

Interconnection

The boopsi gadgets in *boopsi1.c* are not very powerful. The gadgets work independently of each other, forcing the application to unify them. It is possible to make gadget objects update each other without the application's intervention.

Gadgetclass defines two attributes used in this updating process: `ICA_Target` and `ICA_Map`. The `ICA_Target` attribute specifies a pointer to a target object. If an object's attribute changes (and the `OM_NOTIFY` method applies to that attribute), the object sends itself an `OM_NOTIFY` message. If the object has a target object, the target will receive an `OM_UPDATE` message which usually tells the target to set one of its own attributes. For example, when the user slides the knob of a *propgclass* object, its `PGA_Top` attribute changes. If this object has a second *propgclass* object as its target, the second object will receive an `OM_UPDATE` message and will set its corresponding `PGA_Top` attribute to the `PGA_Top` value of the first *propgclass* object.

Because objects of different classes do not always have corresponding attributes, there is a way to map attributes of one object to the attributes of another. The `ICA_Map` accepts a TagItem array of attribute pairs. The first entry in the pair is the source attribute ID and the second is the target object's attribute. For example, an `ICA_Map` that maps a prop gadget's `PGA_Top` attribute to a string gadget's

STRINGA_LongVal attribute would look like this:

```
struct TagItem slidertostring[] = {
    {PGA_Top, STRINGA_LongVal},
    {TAG_END, }
};
```

Note that the OM_UPDATE method has to apply to the target attribute for the change to take place.

Although the gadget attributes ICA_Target and ICA_Map allow boopsi gadgets to update each other, by themselves these attributes do not provide the *application* with any information on changes to the gadgets. Instead of using another gadget as a target object, an object targets an *icclass* object. *Icclass* (or InterConnection) objects are simple information forwarders.

Icclass defines two attributes: ICA_Target and ICA_Map, which are almost identical to the *gadgetclass* attributes. The difference is that *icclass* objects can send the application an IDCMPUPDATE IntuiMessage when it receives an OM_UPDATE. If an *icclass* object receives an OM_UPDATE, it will send the IntuiMessage only if the *icclass* object's ICA_Target is ICTARGET_IDCMP and the updated attribute is mapped to a special dummy attribute, ICSPECIAL_CODE. The IntuiMessage's Code field contains the lower 16 bits of the updated attribute.

Icclass has a more powerful subclass called *modelclass*. A *modelclass* object sends OM_UPDATE messages to an entire list of objects that the *modelclass* object maintains. This makes it possible for gadgets to update each other *and* for the application to find out about the changes. A *modelclass* object "broadcasts" attribute changes to its list of boopsi objects *and* it also lets the application know about attribute changes because it inherits the ICTARGET_IDCMP mechanism from *icclass*. To add objects to a *modelclass* object's list, use the OM_ADDMEMBER method.

The power of *icclass* and *modelclass* lies in using them to create a custom subclass. Consider a group of prop gadgets each of which is used to control one of a color's R, G, B, H, S, and V components. When the user changes one of the color components with a prop gadget, the custom model will be notified of that change and, because it is customized for this specific purpose, it will *recalculate* the values for the remaining components. The model will then let the rest of the prop gadget objects (which are attached to *icclass* objects in the custom model's personal list) know about the changes to their component values so they can move their slider to its new position. All this work is done by the objects themselves; the application does not have to process any of the intermediate input.

In general, boopsi's power lies in its Intuition-supported extensibility. Using the existing boopsi classes as a foundation, you can create entirely new subclasses. This makes it possible to create your own custom gadgetry and have it work perfectly with Intuition, just like any existing gadget. New subclasses of *modelclass* can be used to create gadgets that talk directly to ARexx or the clipboard device. If new classes are general enough, they can be made public so other applications can use them.

```
;/* boopsi1.c - Execute me to compile me with Lattice 5.10a
LC -b1 -cfistq -v -y -j73 boopsi1.c
Blink FROM LIB:c.o,boopsi1.o TO boopsi1 LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/


/* boopsi example showing simple creation of boopsi
** gadgets and using system image:.
*/


#include <intuition/intuition.h>
#include <intuition/classusr.h>
#include <intuition/imageclass.h>
#include <intuition/gadgetclass.h>
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

UBYTE *vers = "\0$VER: boopsi1 1.0";

void ProcessWindow(void);

#define MYPROP       1L
#define MYSTRING     2L
#define MYLEFTBUT    3L
#define MYRIGHTBUT   4L

#define GADTOP      10L
#define STRLEFT     20L
#define STRWIDTH    30L
#define STRHEIGHT   11L
#define INITVALUE   25L
#define PROPWIDTH   100L
#define PROPVIS     10L
#define BUTGADWID   11L
#define BUTGADHEI   11L

struct Screen *screen;
struct Window *window;
struct DrawInfo *drinfo;
struct Library *IntuitionBase;
struct Gadget *prop, *string, *leftbut, *rightbut, *mygadgets;
struct Image *rightbimage, *leftbimage;


void main(void)
{
    struct Gadget *tmpgad;
    WORD gadtop;

    tmpgad = (struct Gadget *)&mygadgets;

    if (IntuitionBase = OpenLibrary("intuition.library", 36L))
    {
        if (screen = LockPubScreen(NULL))
        {
            /* need my screen's DrawInfo for the system gadget images */
            drinfo = GetScreenDrawInfo(screen);

            gadtop = screen->Font->ta_YSize + GADTOP;
            if (window = OpenWindowTags(NULL,
                    WA_Title, (ULONG *)"AMail boopsi1",
                    WA_Height, gadtop+STRHEIGHT+10L,
                    WA_Width,  STRLEFT+STRWIDTH+PROPWIDTH+2*BUTGADWID+15,
                    WA_Flags, WFLG_DEPTHGADGET | WFLG_DRAGBAR | WFLG_CLOSEGADGET,
                    WA_IDCMP, IDCMP_CLOSEWINDOW | IDCMP_GADGETUP,
                            TAG_END))
            {
```

```
            /* Create a boopsi string gadget */
            if (string = (struct Gadget *)NewObject(NULL, "strgclass",
                    /* All the normal Gadget fields */
                    GA_ID,              MYSTRING,
                    GA_Immediate,       TRUE,
                    GA_RelVerify,       TRUE,
                    GA_Top,             gadtop,
                    GA_Left,            STRLEFT,
                    GA_Width,           STRWIDTH,
                    GA_Height,          STRHEIGHT,
                    STRINGA_MaxChars,   3,
                    STRINGA_LongVal,    INITVALUE,
                    STRINGA_Justification, STRINGRIGHT,

                    /* Boopsi makes it easy to link gadgets together.
                    ** The GA_Previous tag accepts a (struct Gadget **)
                    ** to the previous gadget in the list, then changes
                    ** this value to point to the gadget being
                    ** created (in this case, "string").
                    */
                    GA_Previous,        tmpgad,
                    TAG_END))
            {
                /* create the prop gadget */
                if (prop = (struct Gadget *)NewObject(NULL, "propgclass",
                            GA_Immediate,TRUE,
                            GA_RelVerify,TRUE,
                            PGA_Freedom,    FREEHORIZ,
                            PGA_Borderless, FALSE,
                            GA_Left,        STRLEFT + STRWIDTH,
                            GA_Top,         gadtop,
                            GA_Height,      BUTGADHEI,
                            PGA_Top,        INITVALUE,
                            PGA_Visible,    PROPVIS,
                            PGA_Total,      PROPWIDTH,
                            GA_ID,          MYPROP,

                            /* link prop to string and
                            ** make tmpgad point to prop. */
                            GA_Previous,    tmpgad,
                            TAG_END))
                {
                    /* Ask the system for a left arrow image
                    ** for the left arrow gadget. */
                    if (leftbimage = (struct Image *)NewObject(NULL,
                        "sysiclass",

                        /* boopsi needs this screen's DrawInfo
                        ** structure to get the right image.  */
                        SYSIA_DrawInfo,   drinfo,
                        SYSIA_Which,      LEFTIMAGE,

                        /* this will give us 11 x 11 buttons */
                        SYSIA_Size,       SYSISIZE_MEDRES,
                              TAG_END))
                    {
                        /* Now ask for a right arrow */
                        if (rightbimage = (struct Image *)NewObject(NULL,
                            "sysiclass",
                            SYSIA_DrawInfo,   drinfo,
                            SYSIA_Which,      RIGHTIMAGE,
                            SYSIA_Size,       SYSISIZE_MEDRES,
                                        TAG_END))
                        {
                            /* Create the left button */
                            if (leftbut = (struct Gadget *)NewObject(NULL,
                                "buttongclass",
                                GA_ID,          MYLEFTBUT,
                                GA_Immediate, TRUE,
                                GA_RelVerify, TRUE,
                                GA_Image,     leftbimage,
                                GA_Top,       gadtop,
                                GA_Left,      STRLEFT+STRWIDTH+PROPWIDTH-15,
                                GA_Width,     BUTGADWID,
                                GA_Height,    BUTGADHEI,
                                GA_Previous,  tmpgad,
                                TAG_END))
```

```
                        {
                            /* Create the right button */
                            if (rightbut = (struct Gadget *)NewObject(NULL,
                                "buttongclass",
                                GA_ID,          MYRIGHTBUT,
                                GA_Immediate, TRUE,
                                GA_RelVerify, TRUE,
                                GA_Image,       rightbimage,
                                GA_Top,         gadtop,
                                GA_Left,
                                    STRLEFT+STRWIDTH+PROPWIDTH+BUTGADWID-10,
                                GA_Width,       BUTGADWID,
                                GA_Height,      BUTGADHEI,
                                GA_Previous,    tmpgad,
                                TAG_END))
                            {
                                /* All of the gadgets have been created
                                ** and linked together.  Add them to the
                                ** display and display them.
                                */
                                AddGList(window, mygadgets, -1, -1, NULL);
                                RefreshGList(mygadgets, window, NULL, -1);
                                ProcessWindow();
                                RemoveGList(window, mygadgets, -1);
                                DisposeObject(rightbut);
                            }
                            DisposeObject(leftbut);
                        }
                        DisposeObject(rightbimage);
                    }
                    DisposeObject(leftbimage);
                }
                DisposeObject(prop);
            }
            DisposeObject(string);
        }
        CloseWindow(window);
    }
    FreeScreenDrawInfo(screen, drinfo);
    UnlockPubScreen(NULL, screen);
}
CloseLibrary(IntuitionBase);
}
}
```

```
void ProcessWindow(void)
{
    struct IntuiMessage *imsg;
    BOOL returnvalue = TRUE;
    ULONG class;
    LONG currval = INITVALUE;
    struct Gadget *g;


    while (returnvalue)
    {
        WaitPort(window->UserPort);
        while (imsg = (struct IntuiMessage *)GetMsg(window->UserPort))
        {
            g = (struct Gadget *)imsg->IAddress;
            class = imsg->Class;
            ReplyMsg((struct Message *)imsg);
            switch (class)
            {
                case IDCMP_CLOSEWINDOW:
                    returnvalue = FALSE;
                    break;
                case IDCMP_GADGETUP:
                    switch (g->GadgetID)
                    {
                        case MYLEFTBUT:
                            currval--;
                            break;
                        case MYRIGHTBUT:
                            currval++;
                            break;
                        case MYPROP:
                            /* read the prop gadget's value */
                            GetAttr(PGA_TOP, prop, &currval);
                            break;
                        case MYSTRING:
                            /* read the string gadget's value */
                            GetAttr(STRINGA_LongVal, string, &currval);
                            break;
                    }
                    /* make sure the value is between 0 and 90 */
                    if (currval < 0L)
                        currval = 0L;
                    else
                        if (currval > PROPWIDTH - PROPVIS)
                            currval = PROPWIDTH - PROPVIS;

                    /* Update the values of the prop and string.
                    ** gadgets (Intuition takes care of the  refresh). */

                    SetGadgetAttrs( prop, window, NULL,
                                PGA_TOP, currval,
                            TAG_END );

                    SetGadgetAttrs( string, window, NULL,
                           STRINGA_LongVal, currval,
                                TAG_END );

                    break;
            }
        }
    }
}
```

Appendix: Boopsi Class Reference

This appendix documents all the standard classes, their methods, and their attributes. The parameters for each moethod can be found in the appropriate include file. Each attribute is followed by a code that lists the methods that apply to that attribute. The codes are as follows:

I	OM_NEW
S	OM_SET
G	OM_GET
N	OM_NOTIFY
U	OM_UPDATE

Class:	***rootclass***
Superclass:	None
Description:	Base class for all other classes
Include File:	*<intuition/classusr.h>*

New Methods:

OM_NEW - Create a new object. This method is passed a pointer to the "true class" of the object in place of the yet to be created object. The Root class will allocate enough memory for all the instance data of the true class.

OM_DISPOSE - Delete an object. This method will free the same amount of memory as allocated by OM_NEW.

OM_ADDTAIL - Add an object to an Exec list, based on a MinNode in the private *rootclass* object header. You can provide any Exec list you want, and iterate the objects on the list using the Intuition function NextObject().

OM_REMOVE - Remove an object from an Exec list.

The following methods are defined for all objects, but are no-ops in the *rootclass*. Implementation for specific subclasses is done to support these conventions.

OM_ADDMEMBER - Add some object (passed to the method) to the objects list, which is defined by the class. Examples include the broadcast list of *modelclass* objects, and the component gadgets of a *groupgclass* object. This is typically implemented by sending the OM_ADDTAIL message to the member object.

OM_REMMEMBER - Remove a member object previously added by OM_ADDMEMBER. You typically send the OM_REMOVE message to the member object.

OM_SET - Set attributes from passed attribute list. The Root class defines no attributes.

OM_GET - Retrieve an attribute.

OM_UPDATE - "Be updated" of external changes. A no-op for *rootclass*.

OM_NOTIFY - Notify other of changes provided in the message. A no-op for the root class.

Changed Methods:

Not applicable.

Attributes:

None.

Class:	***imageclass***
Superclass:	*rootclass*
Description:	Base class for Intuition Images
Include File:	*<intuition/imageclass.h>*

New Methods:

IM_DRAW - Draw the image in the RastPort passed in the message, at the xoffset and in the state provided in the message.

IM_HITTEST - Return TRUE if the point passed is within the image. This base Image class returns TRUE if the point is within the old Image structure box.

IM_ERASE - Erase an image. This class will call the new graphics library function EraseRect() for its Image box. *IM_ERASE does not test the box dimensions (>= 0).*

IM_MOVE - Erase and redraw an image. This is intended for subclasses capable of performing animation or smooth dragging (e.g., prop gadget knobs). A no-op for *imageclass*.

IM_FRAMEBOX - A no-op for this base class, but implemented by image subclasses which are "stretchable". For example, *frameiclass* objects put an embossed frame around a text label or glyph. The message passed to this method contains a pointer to a box describing the dimensions of the contents to frame, and a pointer to a frame box for the result. There is also a FrameFlags field which currentlly has only one defined flag, FRAMEF_SPECIFY.

The idea behind this frame business is to support multiple gadgets (as found in a system requester) which share a single frame image object, but render it in different dimensions appropriate for their enclosed label. The operation of this method should be defined by suitable subclasses as follows, and return non-zero to indicate that they support this method.

If FRAMEF_SPECIFY is not set, you should set up the povided frame box with the position and dimensions of the frame you would render to enclose the contents box. This is an inquiry function used, for example, by a gadget which wants to know what sized frame dimensions to pass to IM_DRAWFRAME (defined below) and how to center the contents.

The caller might then add a little aesthetic margin by increasing the dimensions (or might decide to make a column of framed objects all the same width as the largest one), and having done so, can call the method IM_FRAMEBOX again, this time with FRAMEF_SPECIFY set. Now, your method is expected to respect the dimensions specified in the frame box (even if you think they are too small!) and to set up only the position of the frame box appropriately for enclosing the provided contents box. Currently implemented frame image classes always center the contents, but future classes might want to support an attribute for optional left or right justification of the contents.

IM_DRAWFRAME - Image subclasses which assign proper meaning to this class (i.e. *frameiclass*) should respect the dimensions that are passed in this message by superceding this method. If *imageclass* sees this message, it will convert it to an IM_DRAW message and send it back to the image's true class. This way, any subclass of *imageclass* which doesn't implement or recognize the IM_DRAWFRAME method will default to drawing objects at their "natural" dimensions.

IM_HITFRAME - Perform a hit test for the image respecting the frame dimensions passed in the message. Useful for something like a stretchable rounded box. This class just performs the same as IM_HITTEST, ignoring the dimensions passed.

IM_ERASEFRAME - Erase an image respecting the frame dimensions passed in the message. This class calls EraseRect() for the frame dimensions, again not checking dimension validity.

Changed Methods:

OM_NEW - Instance data contains an Image structure, and its Depth field is initialized to CUSTOMIMAGEDEPTH, which identifies such images to Intuition. The Width and Height fields are set to arbitrary positive numbers for safety, but you should always set them to something meaningful.

OM_SET - Applies all supported attributes, returns '1'.

Attributes:

IA_Left	(ISG)
IA_Top	(ISG)
IA_Width	(ISG)
IA_Height	(ISG)

These attributes are stored (by *imageclass*) as values in the familiar Image structure.

IA_FGPen	(ISG)
IA_BGPen	(ISG)

These attributes are maintained in the PlanePick and PlaneOnOff fields, which concept they generalize.

IA_Data (ISG)
A pointer to general image data. This value is stored in the ImageData field of the old Image structure.

IA_LineWidth ()
A no-op for *imageclass*.

IA_Pens ()
Defined but not supported by *imageclass*. Pointer to UWORD pens[], terminated by ~0. This can be used to the exclusion of DrawInfo attributes or parameters, or as an override.

Class: *frameiclass*
Superclass: *imageclass*
Description: A raised, sunken, or chisled image class supporting frame dimensions and Intuition new look parameters.
Include File: *<intuition/imageclass.h>*

New Methods:

None, but implements some of the no-ops in *imageclass*.

Changed Methods:

OM_GET - A no-op for this class.

IM_DRAW - Draws the embossed frame in its natural (Image) dimensions. Frame thickness support of IA_LineWidth is currently not implemented. Frame type is determined by attributes below, colors determined by DrawInfo and supports these drawing states:

IDS_NORMAL, IDS_INACTIVENORMAL, IDS_DISABLED:
Renders edges in shadowPen, shinePen, and backgroundPen for the interior. NOTE: No ghosting is done for IDS_DISABLED.

IDS_SELECTED, IDS_INACTIVESELECTED:
Renders edges in shadowPen, shinePen, and hifillPen, for the interior.

IM_DRAWFRAME - Same as IM_DRAW, but draws frame to specified exterior dimensions.

IM_FRAMEBOX - Centers contents, adds dimension for frame thickness, plus an interior margin of the same thickness.

Attributes:

IA_Recessed (IS)
Specifies that the frame should be recessed into the drawing surface (shadowPen used on upper and left edges). Default is FALSE, a raised frame.

IA_DoubleEmboss (IS)
Two nested embossed frames for "chisled" boxes or "ridges". Not currently supported.

IA_EdgesOnly (IS)
Does not fill frame, just draws the edges.

IA_LineWidth 0
Not currently supported.

Class: ***sysiclass***
Superclass: ***imageclass***
Description: Class for system and standard application images.
Include File: ***<intuition/imageclass.h>***

New Methods:

None.

Changed Methods:

`OM_NEW` - Executes a drawing commands list to create a fast-rendering image object. The glyph of the object is defined by the attribute `SYSIA_Which`. The `SYSIA_DrawInfo` parameter is ***required*** at initialization, as is `IA_Height`. Image size defaults are established by `SYSIA_Size`, but some images (title bar gadget images) require `IA_Height`. Images are scaled to their dimensions.

Attributes:

`SYSIA_DrawInfo` (IS)
This attribute must be passed to `OM_NEW` and `OM_SET` to allow the image to be generated into a bitmap cache.

`SYSIA_Which` (I)
Identifies which of the system image glyphs the caller wants. It can be one of the following:

`DEPTHIMAGE`	Window depth arrangement image.
`ZOOMIMAGE`	Window Zoom image.
`SIZEIMAGE`	Window Sizing image.
`CLOSEIMAGE`	Window close image.
`SDEPTHIMAGE`	Screen depth arrangement image.
`LEFTIMAGE`	Left arrow image.
`UPIMAGE`	Up arrow image.
`RIGHTIMAGE`	Right arrow image.
`DOWNIMAGE`	Down arrow image.
`CHECKIMAGE`	Checkmark image.
`MXIMAGE`	Radio button image.

`SYSIA_Size` (I)
Identifies which default dimensions to use for the object. This generalizes Intuition's older concept of two different system image dimensions. The system currently uses `SYSISIZE_MEDRES` as the default, `SYSISIZE_HIRES` for screens with title bars greater than 22 pixels, and `SYSISIZE_LORES` for screens with fat pixels.

Class: ***fillrectclass***
Superclass: ***imageclass***
Description: Rectangle with frame and pattern support
Include File: ***<intuition/imageclass.h>***

New Methods:

None.

Changed Methods:

`IM_DRAW` - Draws the pattern in its natural (Image) dimensions.

`IM_DRAWFRAME` - Same as `IM_DRAW`, but draws pattern to specified exterior dimensions.

Attributes:

`IA_APattern` (IS)
Points to the pattern to fill in the Image structure's ImageData field.

`IA_APatSize` (IS)
The pattern's dimension.

`IA_Mode` (IS)
Drawing mode for the pattern.

Class: ***itexticlass***
Superclass: ***imageclass***
Description: IntuiText equivalent with attribute override.
Include File: ***<intuition/imageclass.h>***

New Methods:

None.

Changed Methods:

`IM_DRAW` - Draws IntuiText specified as `IA_Data` overriding the attributes specified in the IntuiText structure. The Mode used is JAM1, the color as specified as `IA_FGPen`. The position of the itexticlass object is added to the position of the IntuiText.

This class was defined to make new-looking AutoRequesters without changing the IntuiText structures passed to it.

Attributes:

`IA_Data` (ISG)
Processed blindly by superclass, must be a pointer to a normal IntuiText structure.

`IA_FGPen` (ISG)
Processed by superclass, used as APen when drawing text.

`IA_Left, IA_Top` (ISG)
Added to IntuiText offsets.

Class: ***icclass***
Superclass: *rootclass*
Description: Base class of simple forwarding interconnection.
Include File: *<intuition/icclass.h>*

New Methods:

`ICM_SETLOOP`
`ICM_CLEARLOOP`
`ICM_CHECKLOOP` - All of these methods are used internally by subclasses to manage a loop inhibition for broadcasted messages. They increment, decrement, and return the value of that counter.

Changed Methods:

`OM_SET` - Sets attributes and returns 0. Note that this is *not* the same behavior as `OM_NOTIFY`.

`OM_NOTIFY` - Issues an `OM_UPDATE` message to the object indicated by attribute `ICA_Target`, first converting the attribute tags according to the `ICA_Map` attribute. The internal loop count is incremented until the `OM_UPDATE` method it invokes on `ICA_Target` returns. Also, it will do nothing at all if that loop count was non-zero to begin with, preventing itself from participating in an infinite loop.

`OM_UPDATE` - Receive update notices from others. This base IC class just passes the message along, treating it exactly like an `OM_NOTIFY` message. Subclasses will probably redefine this method to perform a calculation, the result of which they might forward to others by issuing themselves an `OM_NOTIFY` message. Unfortunately, this class *converts* the `OM_SET` and `OM_NOTIFY` messages to an `OM_NOTIFY` by changing their MethodID field, and does not restore it to what it originally was.

Attributes:

`ICA_Target` (IS)
Target object for `OM_UPDATE` messages. If this attribute is given the value `ICTARGET_IDCMP`, then the notification will consist of sending an IDCMPUPDATE IntuiMessage to a window. The window is determined by the GadgetInfo structure passed around when the object is connected to gadgets.

`ICA_Map` (IS)
Attribute mapping list.

`ICSPECIAL_CODE` (*)
This is a magic dummy attribute: if it occurs as a target in the `ICA_Map` mapping list, and `ICA_Target` is `ICTARGET_IDCMP`, then the new value of the corresponding attribute will be copied into the IntuiMessage.Code field of the IDCMPUPDATE message (just the lower sixteen bits of the attribute value will fit). This sometimes makes it particularly simple to process IDCMPUPDATE messages with a single item of interest.

Class: ***modelclass***
Superclass: *icclass*
Description: Provides "broadcast" interconnection.
Include File: *<intuition/icclass.h>*

New Methods:

None.

Changed Methods:

`OM_ADDMEMBER` - Adds an object to the the Model's internally maintained broadcast list. ***Anything on this list will be DISPOSED when the model object is disposed.***

`OM_REMMEMBER` - Removes objects added by `OM_ADDMEMBER`.

`OM_DISPOSE` - Disposes members of the broadcast list as well as itself.

`OM_NOTIFY`, `OM_UPDATE` - these behave like the *icclass* methods, but first an `OM_UPDATE` message is sent to all members of the object's broadcast list, un-mapped. The members of the broadcast list are typically icclass objects, which have gadgets as their targets and appropriate mapping lists. If you are creating a useful subclass, you will probably want to intercept `OM_UPDATE`, but pass `OM_NOTIFY` to *modelclass*, your superclass.

Attributes:

`ICA_Map`, `ICA_Target` - Same as for the superclass (*icclass*).

Class: ***gadgetclass***
Superclass: *rootclass*
Description: Base class for Intuition compatible gadget classes
Include File: *<intuition/gadgetclass.h>*

New Methods:

Intuition uses the following methods to control a gadget. Only custom gadget class implentors will be interested in these.

```
GM_HITTEST
GM_RENDER
GM_GOACTIVE
GM_HANDLEINPUT
GM_GOINACTIVE
```

Changed Methods:

`OM_NEW` - Initializes transparent Gadget structure. LeftEdge and TopEdge are set to 0, Width and Height are arbitrary constants which you should override. Sets GadgetType to CUSTOMGADGET, and installs a pointer to the *gadgetclass* data structure in the MutualExclude field. These objects are also valid "custom gadgets."

Links self into a NextGadget linked list if `GA_Previous` is passed to `OM_NEW`.

`OM_UPDATE` and `OM_SET` are used simply to change the attributes. This *gadgetclass* does not implement any concrete gadgets. This is left to the subclasses.

`OM_NOTIFY` - This class will issue an `OM_UPDATE` message to `ICA_Target` through `ICA_Map`. Subclasses of *gadgetclass* are advised to let the superclass handle this method.

Attributes:

`GA_Left` (IS)
`GA_Top` (IS)
`GA_Width` (IS)
`GA_Height` (IS)
These correspond to position and dimension fields in the Intuition Gadget structure. Setting these *clears* the "gadget relative" flags (below).

`GA_RelRight` (IS)
`GA_RelBottom` (IS)
`GA_RelWidth` (IS)
`GA_RelHeight` (IS)
These are alternative position/dimension attributes. Setting these stores the corresponding data in the Left/Top/Width/Height Gadget fields, respectively, and sets the corresponding "relative" flag (`GRELRIGHT`, `GRELBOTTOM`, and so on).

`GA_IntuiText` (IS)
`GA_Text` (IS)
`GA_LabelImage` (IS)
This copies the ti_Data value blindly into the Gadget.GadgetText field, and sets the flags `LABELSTRING` and `LABELIMAGE` as appropriate in Gadget.Flags. `GA_IntuiText` requires that ti_Data be an IntuiText pointer, as with old-style gadgets. `GA_Text` takes a pointer to a `NULL`-terminated string (UBYTE *). `GA_LabelImage` takes a pointer to a (boopsi) image.

Classes which support the attributes other than `GA_IntuiText` must document themselves appropriately, but this facility allows us to have low-overhead text gadgets and gadgets with iconic labels.

`GA_Image` (IS)
This may be a boopsi image or a regular image. Things are designed so that the image may be shared between gadgets. The image will *not* be disposed when the gadget object is disposed.

`GA_Border` (IS)
`GA_SelectRender` (IS)
`GA_ID` (IS)
`GA_UserData` (IS)
`GA_SpecialInfo` (IS)
All of this group of attributes correspond to fields in the Gadget structure.

`GA_Disabled` (IS)
`GA_GZZGadget` (IS)
`GA_Selected` (IS)
`GA_EndGadget` (IS)
`GA_Immediate` (IS)
`GA_RelVerify` (IS)
`GA_FollowMouse` (IS)
`GA_RightBorder` (IS)
`GA_LeftBorder` (IS)
`GA_TopBorder` (IS)
`GA_BottomBorder` (IS)
`GA_ToggleSelect` (IS)
`GA_SysGadget` (IS)
`GA_TabCycle` (IS)
This group corresponds to Boolean attributes (flags) in the Gadget structure. This class will put the correct flag in the correct field. You can pass any non-zero value in the ti_Data tag field to cause the corresponding flag to be set. Pass zero to clear.

`GA_HighLight` (IS)
Sets the "`GADGHIGHBITS`" portion of Gadget.Flags to the attribute value in ti_Data.

`GA_SysGType` (IS)
Sets the system gadget type portion of Gadget.GadgetType to the (masked) value in ti_Data.

`GA_Previous` (I)
Previous gadget (or (struct Gadget **)) in linked list. This attribute cannot be used to link new gadgets into the gadget list of an open window or requester. You must use AddGList().

`ICA_Target` (IS)
`ICA_Map` (IS)
These are given the same meaning as in *icclass*, from which they are borrowed.

Class: *propgclass*
Superclass: *gadgetclass*
Description: Extended function proportional gadgets.
Include File: *<intuition/gadgetclass.h>*

New Methods:

None.

Changed Methods:

All methods defined by *gadgetclass* are handled to provide compatible proportional gadget processing.

OM_SET, OM_UPDATE - Changes attributes and, if needed and if propgclass is the true class, will update the slider visuals by updating the knob position and dimensions.

GM_HANDLEINPUT - If the knob position changes sufficiently to change PGA_Top, this method will issue an OM_NOTIFY message, with attributes PGA_Top and GA_ID. The OPUF_INTERIM flag will be set for intermediate messages issued while the mouse is dragging the slider knob. This method Will issue a message with OPUF_INTERIM clear when it is done, which is the "final value".

Attributes:

GA_Image (I)
GA_Border (I)
If you don't pass GA_Image to NewObject(), the gadget will create and use an AUTOKNOB. Likewise if you pass (the "illegal" attribute) GA_Border: an AUTOKNOB will be used instead.

GA_HighLight (I)
GADGHBOX highlighting is not allowed, and will be converted to GADGHBOX.

GA_SpecialInfo ()
This attributed is "forced" to point to the PropInfo allocated for objects of this class.

Other *gadgetclass* attributes are passed along to the superclass.

PGA_Freedom (IG)
May be either of FREEHORIZ or FREEVERT (not both). The default is FREEVERT.

PGA_Borderless (I)
Means the same as PropInfo.Flags BORDERLESS.

PGA_HorizPot, PGA_VertPot, PGA_HorizBody, PGA_VertBody
These are defined in the include file but obsolete and will not be supported. Class propgclass supports gadgets that are vertical or horizontally free, but not both.

PGA_Top (ISGNU)
PGA_Visible (ISU)
PGA_Total (ISU)
These attributes are very useful replacements to Pot and Body variables. They are based on the use of proportional gadgets to control scrolling text. When scrolling 100 lines of text in a 25 line visible window, you would set PGA_Total to 100, PGA_Visible to 25, and watch PGA_Top run from 0 to 75 (the top line of the last page). All internal prop gadget values will be calculated based on these (depending on whether the gadget is FREEHORIZ or FREEVERT). "Container clicks" for page jumps will leave an overlap of one line, unless the value PGA_Visible is 1, in which case the prop gadget acts as an integer numeric slider taking values from 0 to PGA_Total - 1. Note that PGA_Top has OM_NOTIFY access. All three of these attributes have OM_UPDATE access, so they can be controlled via interconnections.

PGA_NewLook (I)
Equivalent to PropInfo.Flags PROPNEWLOOK flag.

Class: *strgclass*
Superclass: *gadgetclass*
Description: Intuition compatible string gadgets.
Include File: *<intuition/gadgetclass.h>*

New Methods:

None.

Changed Methods:

All methods defined by *gadgetclass* are handled to provide compatible string gadget processing.

OM_NEW - Sets up StringInfo and StringExtend structures. Will allocate a Buffer if needed and will use shared data buffers for UndoBuffer and WorkBuffer if the MaxChars is less than SG_DEFAULTMAXCHARS (128). Default text pens are FG = 1, BG = 0;

Attributes:

GA_ID (ISG)
Will be included in OM_NOTIFY messages generated.

STRINGA_MaxChars (I)
STRINGA_Buffer (I)
STRINGA_UndoBuffer (I)
STRINGA_WorkBuffer (I)
Specify various buffers defined for string gadgets and extended string gadgets. If your value of STRINGA_MaxChars is less than SG_DEFAULTMAXCHARS (120 for now), then this class can provide all these buffers for you. Note that UndoBuffer and WorkBuffer can be shared by many separate gadgets, providing they are as large as the largest MaxChars they will encounter.

STRINGA_BufferPos (ISU)
STRINGA_DispPos (ISU)
Familiar cursor and scroll position.

STRINGA_AltKeyMap (IS)
Same as StringInfo.AltKeyMap.

STRINGA_Font (IS)
Font for string gadget text. Must be an open TextFont.

STRINGA_Pens (IS)
Pen numbers, packed as two shorts into a longword, for rendering gadget text.

STRINGA_ActivePens (IS)
Optional pen numbers, packed as two shorts into a longword, for rendering gadget text, when the gadget is active.

STRINGA_EditHook (I)
Custom string gadget edit hook.

STRINGA_EditModes (IS)
Value taken from flags defined in *<intuition/sghooks.h>* for initial editing modes.

STRINGA_ReplaceMode (IS)
STRINGA_FixedFieldMode (IS)
STRINGA_NoFilterMode (IS)
These three are independent Boolean equivalents to the individual flags that you can set for STRINGA_EditModes.

STRINGA_Justification (IS)
Takes the values STRINGCENTER, STRINGRIGHT, and STRINGLEFT (which is 0).

STRINGA_LongVal (ISGNU)
When you specify this to a string gadget object, it means first that the gadget is now for integer entry only, and the value of the gadget takes the numeric value passed in ti_Data. Note that this attribute has OM_NOTIFY and OM_UPDATE access.

STRINGA_TextVal (ISGNU)
When you specify this to a string gadget object, it means that the gadget is for text entry only, and the value of the gadget is copied from the string value passed as a UBYTE pointer in ti_Data. Note that this attribute has OM_NOTIFY and OM_UPDATE access.

STRINGA_ExitHelp (IS)
Set this if you want the gadget to exit when the "Help" key is pressed. Look for a code of 0x5f, the rawkey code for help.

Notification messages will be issued whenever the gadget chooses to go inactive (not when it is aborted). The OPUF_INTERIM flag is always clear.

Class: *groupgclass*
Superclass: *gadgetclass*
Description: Composite gadget objects
Include File: *<intuition/gadgetclass.h>*

New Methods:

None

Changed Methods:

OM_SET - Passes most attributes to the superclass, but propagates changes in position to its members appropriately. Also, GA_Width and GA_Height are calculated from the position and dimension of the membership.

OM_ADDMEMBER - add a gadget to the group list. Add the gadgets you want in the group right after you create it and leave them there until you are done. Note that all members of the *groupgclass* object will be deleted by OM_DISPOSE.

OM_DISPOSE - this class will dispose of the object and all its member gadgets.

Attributes:

LAYOUTA_Orientation
LAYOUTA_Spacing
LAYOUTA_LayoutObj
These attributes specify simple layout parameters and a "layout object delegate" are not currently implemented. GA_Width and GA_Height are calculated from the membership.

GA_Left (IS)
GA_Position (IS)
These are propagated magically to the membership.

The gadget relative flags (GA_RelWidth, GA_RelHeight, and so on) are not supported.

Class: *buttongclass*
Superclass: *gadgetclass*
Description: A (repeatable) command button gadget.
Include File: *<intuition/gadgetclass.h>*

New Methods:

None.

Changed Methods:

GM_HITTEST - Delegates this question to its GA_Image attribute.

GM_HANDLEINPUT - Will behave like a button, but continuously issues OM_NOTIFY messages for each IECLASS_TIMER event.

Flag OPUF_INTERIM will be set for all but the last notification.

The notified attribute is GA_ID, with a twist: the value sent will be the *negative* of GadgetID if the pointer is not currently over the gadget image.

GM_RENDER - All rendering is passed along to the GadgetRender Image. The state provided is IDS_INACTIVESELECTED, IDS_INACTIVENORMAL, IDS_SELECTED, or IDS_NORMAL.

Attributes:

GA_Image (IS)
Changing the image will cause the gadget to refresh itself.

Class: *frbuttonclass*
Superclass: *buttongclass*
Description: A button gadget that knows how to outline its "label" within a shared "frame image".

New Methods:

None.

Changed Methods:

OM_NEW - Will set up its dimensions depending on GA_Image, including support for frame images. If GA_Image understands the IM_FRAMEBOX method, dimensions are calculated to surround the "label" stashed in GadgetText, which can be GA_IntuiText, GA_Text, or GA_LabelImage.

OM_SET - If you change the dimensions, this method will adjust the contents by using IM_FRAMEBOX with FRAMEIF_SPECIFY.

GM_HITTEST - uses IM_HITFRAME.

GM_RENDER - uses IM_DRAWFRAME. First draws the frame, then draws the contents or label described under OM_NEW.

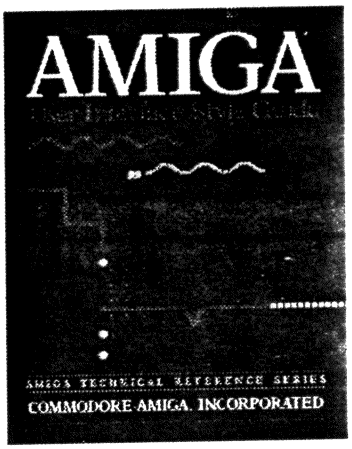
Attributes:

GA_Width (IS)
GA_Height (IS)
If you change these, the contents will be readjusted and the gadget re-rendered.

Amiga Mail
Volume II
March/April 1991

From the Desk of Jeff Scherb

The Amiga is receiving more attention from professional users these days, which makes the user interface issue even more important. Amiga applications are now competing with Mac and Windows applications. The vast majority of applications for those platforms conform to guidelines initially set down by the respective manufacturer. As a result, it is relatively easy for a Windows spreadsheet user to learn how to use a Windows word processor because the user knows how to operate the user interface. The commands and menus are already familiar.



The new *Amiga User Interface Style Guide* describes the standards and conventions for the three Amiga user interfaces -- the GUI, the CLI, and ARexx. This book took over a year to develop and is the result of months of research into existing user interface conventions on a variety of platforms. CATS coordinated with several key developers to help form a set of practical and reasonable guidelines for the Amiga user interface.

You might wonder what the drawbacks are for not following user interface standards. No, the Interface Police are not going to show up at your door one morning -- the drawbacks are much more subtle than that. Not following commonly accepted practices makes an application's user interface less predictable, making it harder for the user to learn how to use the application. If the application's interface is too unpredictable, within a few minutes the user may give up trying to use the application. This leads to lower market acceptance -- one penalty of not following user interface standards.

Another disadvantage of not following the user interface standard is that it increases the amount of code a developer has to maintain. Release 2.0 provides some standard GUI elements through standard system libraries (such as the standard requesters of *asl.library*). Not using these standard libraries will produce unconventional (and possibly confusing) GUI elements that you will have to update and correct yourself, rather than system standard GUI elements that Commodore maintains as part of the Amiga system software.

The Style Guide is part of the new Release 2.0 ROM Kernel Reference series published by Addison-Wesley. The book should be available by the time you read this. Developers will be able to get the Style Guide through CATS. The book should also be available in bookstores. At a retail price of $21.95, everyone involved in creating applications for the Amiga should have one.

CATS has several other development aids in the works. We're working on a user interface design, protoyping, and code generation tool which should be available by DevCon in September. Currently known as *AppBuilder*, this tool allows you to quickly build an interface which follows the guidelines set forth in the Style Guide. After you're satisfied with the look and feel of the interface, *AppBuilder* can generate C or assembler source code. The code can be compiled and linked with the *AppShell*, which provides standard event processing code. With the combination of *AppBuilder* and *AppShell*, you'll be able to spend your development time on the functionality of your application instead of on gadget creation and housekeeping.

The Amiga is growing up. It is finding its way into more professional markets, and increasingly competing with and being compared to the Mac OS and Windows. I encourage all of you to review your applications in light of the information in the Guide and plan for interface improvements in your next release.

-JS

Table of Contents

March/April 1991

I
Programming Practices and Standards
general purpose Amiga programming

Tag Items and Tag Lists . . . . . I - 1
staff

2.0 Version Strings . . . . . I - 3
Carolyn Scheppner

II
AmigaDOS
dos.library

Executing External Programs with Release 2.0 . . . . . II - 1
Carolyn Scheppner

AmigaDOS Packet Interface Specification . . . . . II - 5
John Toebes

III
Exec
the multitasking executive, exec.library

Using SetFunction() in a Debugger . . . . . III - 1
Ewout Walraven

IV
Intuition and Workbench
the Amiga user interface

Obtaining an Icon Image . . . . . IV - 1
David Junod

An Introduction to V36 Screens and Windows . . . . . IV - 3
Ewout Walraven

Opening Screens and Windows on Any Amiga . . . . . IV - 17
Ewout Walraven

Introduction to Boopsi . . . . . IV - 23
John Orr

V
Graphics
animation, text, the Blitter, layers.library

Using the ECS Genlocking Features . . . . . **V - 1**
Ewout Walraven

VI
Sound and Music
audio, speech, MIDI

VII
IFF
iffparse.library and other IFF topics

The DR2D FORM . . . . . **VII - 1**
Ross Cunniff and John Orr

VIII
File Systems, Handlers, and Storage Devices
DOS packet handlers, storage media, SCSI

IX
Timer, Serial, and Parallel
timer.device, serial port, parallel port

X
Output
printers, printing, Postscript

XI
Input
user input, Commodities Exchange

Introduction to Commodities Exchange . . . . . **XI - 1**
John Orr

For example, the input description string "rawkey -caps -lalt -relativemouse -upstroke ralt tab" matches a tab upstroke or downstroke with the right alt key pressed whether or not the left alt, either shift, or the capslock keys are down. The following IX structure corresponds to that input description string:

```
IX ix = {
    IX_VERSION,                     /* The version */
    IECLASS_RAWKEY,                 /* We're looking for a RAWKEY event */
    0x42,                           /* The key the usa0 keymap maps to a tab */
    0x00FF & (~IECODE_UP_PREFIX),   /* We want up and down key presses */
    IEQUALIFIER_RALT,               /* The right alt key must be down */
    0xFFFF & ~(IEQUALIFIER_LALT | IEQUALIFIER_LSHIFT |
        IEQUALIFIER_RSHIFT | IEQUALIFIER_CAPSLOCK | IEQUALIFIER_RELATIVEMOUSE),
        /* don't care about left alt, shift, capslock, or relativemouse qualifiers */
    IXSYM_CAPS  /* The shift keys and the capslock key qualifers are all equivalent */
};
```

The CxFilter() macro only accepts a description string to describe an input event. A commodity can change this filter, however.

```
void SetFilter( CxObj *filter, char *descrstring );
void SetFilterIX( CxObj *filter, IX *ix );
```

SetFilter() and SetFilterIX() change which input events a filter CxObject diverts. SetFilter() accepts a pointer to an input description string. SetFilterIX() accepts a pointer to an IX input expression. A commodity that uses either of these functions should check the filter's error code with CxObjError() to make sure the change worked.

```
errorcode = LONG ParseIX( char *descrstring, IX *ix );
```

The function ParseIX() parses an input description string and translates it into an IX input expression. Commodities Exchange uses ParseIX() to convert the description string in CxFilter() to an IX input expression. As was mentioned previously, as of *commodities.library* version 37.3, ParseIX() does not work with certain kinds of input strings.

This article is by no means an exhaustive description of Commodities Exchange. For more information, see the Commodities Exchange Autodocs (*commodities.doc* for *commodities.library* and *cx.doc* for the macros and linker library functions) and the sample commodities from the 1990 Atlanta DevCon disks.

```
/* broker.c - Simple skeletal example of opening a broker
 * compiled with SASC 5.10
 * LC -b0 -cfist -v broker.c
 * Blink FROM LIB:c.o,broker.o TO broker LIBRARY
 * LIB:LC.lib,LIB:Amiga.lib
 */
#include <exec/libraries.h>
#include <libraries/commodities.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/alib_commodities_protos.h>
#include <clib/commodities_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

struct Library *CxBase;

void main(void);
LONG ProcessMsg(void);

CxObj *broker;

struct MsgPort *broker_mp;

struct NewBroker newbroker = {
    NB_VERSION, /* nb_Version - Version of the NewBroker structure */
    "AmigaMail broker", /* nb_Name - Name CX uses to identify this commodity */
    "Broker",   /* nb_Title - Title of commodity that appears in CXExchange */
    "A simple example of a broker",  /* nb_Descr - Description of the commodity */
    0,          /* nb_Unique - Tells CX not to launch new commodity with same name */
    0,          /* nb_Flags - Tells CX if this commodity has a window */
    0,          /* nb_Pri - This commodity's priority */
    0,          /* nb_Port - MsgPort CX talks to */
    0           /* nb_ReservedChannel - reserved for later use */
};

ULONG cxsigflag;

void main()
{
    /* Before bothering with anything else, open the library */
    if (CxBase = OpenLibrary("commodities.library", 37L))
    {
        /* Commodities talks to a Commodities application through
         * an Exec Message port, which the application provides
         */
        if (broker_mp = CreateMsgPort())
        {
            newbroker.nb_Port = broker_mp;

            /* The commodities.library function CxBroker() adds a
             * broker to the master list.  It takes two arguments,
             * a pointer to a NewBroker structure and a pointer to
             * a LONG.  The NewBroker structure contains information
             * to set up the broker.  If the second argument is not
             * NULL, CxBroker will fill it in with an error code.
             */
            if (broker = CxBroker(&newbroker, NULL))
            {
                cxsigflag = 1L << broker_mp->mp_SigBit;

                /* After it's set up correctly, the broker
                 * has to be activated
                 */
                ActivateCxObj(broker, 1L);

                /* the main processing loop */
                while (ProcessMsg());

                /* It's time to clean up.  Start by removing
                 * the broker from the Commodities master list.
                 * The DeleteCxObjAll() function will take care
                 * of removing a CxObject and all those connected
                 * to it from the Commodities network
                 */
                DeleteCxObj(broker);
            }
            DeleteMsgPort(broker_mp);
        }
        CloseLibrary(CxBase);
    }
}

LONG ProcessMsg(void)
{
    CxMsg *msg;

    ULONG sigrcvd, msgid, msgtype;
    LONG returnvalue = 1L;

    /* wait for something to happen */
    sigrcvd = Wait(SIGBREAKF_CTRL_C | cxsigflag);

    /* process any messages */
    while(msg = (CxMsg *)GetMsg(broker_mp))
    {
        /* Extract necessary information from the CxMessage and return it */
        msgid = CxMsgID(msg);
        msgtype = CxMsgType(msg);
        ReplyMsg((struct Message *)msg);

        switch(msgtype)
        {
            case CXM_IEVENT:
                /* Shouldn't get any of these in this example */
                break;
            case CXM_COMMAND:
                /* Commodities has sent a command */
                printf("A command: ");
                switch(msgid)
                {
                    case CXCMD_DISABLE:
                        printf("CXCMD_DISABLE\n");
                        /* The user clicked Commodities Exchange disable
                         * gadget better disable
                         */
                        ActivateCxObj(broker, 0L);
                        break;
                    case CXCMD_ENABLE:
                        /* user clicked enable gadget */
                        printf("CXCMD_ENABLE\n");
                        ActivateCxObj(broker, 1L);
                        break;
                    case CXCMD_KILL:
                        /* user clicked kill gadget, better quit */
                        printf("CXCMD_KILL\n");
                        returnvalue = 0L;
                        break;
                }
                break;
            default:
                printf("Unknown msgtype\n");
                break;
        }
    }

    /* Test to see if user tried to break */
    if (sigrcvd & SIGBREAKF_CTRL_C)
    {
        returnvalue = 0L;
        printf("CTRL C signal break\n");
    }
    return(returnvalue);
}
```

```
/* HotKey.c - Simple hot key commodity
 * compiled with SASC 5.10
 * LC -b0 -cfist -v hotkey.c
 * Blink FROM LIB:c.o,hotkey.o TO hotkey LIBRARY LIB:LC.lib,LIB:Amiga.lib
 */
#include <exec/libraries.h>
#include <libraries/commodities.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/alib_commodities_protos.h>
#include <clib/commodities_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

#define EVT_HOTKEY 1L

struct Library *CxBase, *IconBase;

void main(int, char **);
LONG ProcessMsg(void);

struct MsgPort *broker_mp;

CxObj *broker, *filter, *sender, *translate;

struct NewBroker newbroker = {
    NB_VERSION,
    "AmigaMail HotKey",          /* string to identify this broker */
    "A Simple HotKey",
    "A simple hot key commodity",
    NBU_UNIQUE | NBU_NOTIFY,     /* Don't want any new commodities
                                  * starting with this name.  If someone
                                  * tries it, let me know */
    0,
    0,
    0,
    0
};


ULONG cxsigflag;

void main(int argc, char **argv)
{
    char *hotkey, **ttypes;

    if (CxBase = OpenLibrary("commodities.library", 37L))
    {
        /* open the icon.library for the support library
         * functions, ArgArrayInit() and ArgArrayDone()
         */
        if (IconBase = OpenLibrary("icon.library", 36L))
        {
            if (broker_mp = CreateMsgPort())
            {
                newbroker.nb_Port = broker_mp;
                cxsigflag = 1L << broker_mp->mp_SigBit;

                /* ArgArrayInit() is a support library function
                 * (from the 2.0 version of amiga.lib) that makes it
                 * easy to read arguments from either a CLI or from
                 * the Workbench's ToolTypes.  Because it uses
                 * icon.library, the library has to be open before
                 * calling this function.  ArgArrayDone() cleans up
                 * after this function.
                 */
                ttypes = ArgArrayInit(argc, argv);

                /* ArgInt() (also from amiga.lib) searches through the
                 * array set up by ArgArrayInit() for a specific
                 * ToolType.  If it finds one, it returns the numeric
                 * value of the number that followed the ToolType
                 * (i.e. CX_PRIORITY=7).  If it doesn't find the ToolType,
                 * it returns the default value (the third argument)
                 */
                newbroker.nb_Pri = (BYTE)ArgInt(ttypes, "CX_PRIORITY", 0);

                /* ArgString() works just like ArgInt(), except it
                 * returns a pointer to a string rather than an integer.
                 * In the example below, if there is no ToolType "HOTKEY",
                 * the function returns a pointer to "rawkey control esc".
                 */
                hotkey = ArgString(ttypes, "HOTKEY", "rawkey control esc");

                if (broker = CxBroker(&newbroker, NULL))
                {
                    /* CxFilter() is a macro that creates a filter
                     * CxObject.  This filter passes input events that
                     * match the string pointed to by hotkey.
                     */
                    if (filter = CxFilter(hotkey))
                    {
                        /* Add a CxObject to another's personal list */
                        AttachCxObj(broker, filter);

                        /* CxSender() creates a sender CxObject.  Every
                         * time a sender gets a CxMessage, it sends a new
                         * CxMessage to the port pointed to in the first
                         * argument.  CxSender()'s second argument will be
                         * the ID of any CxMessages the sender sends to
                         * the port.  The data pointer associated with the
                         * CxMessage will point to a *COPY* of the
                         * InputEvent structure associated with the orginal
                         * CxMessage.
                         */
                        if (sender = CxSender(broker_mp, EVT_HOTKEY))
                        {
                            AttachCxObj(filter, sender);

                            /* CxTranslate() creates a translate CxObject.
                             * When a translate CxObject gets a CxMessage,
                             * it deletes the original CxMessage and adds
                             * a new input event to the input.device's
                             * input stream after the Commodities input
                             * handler.  CxTranslate's argument points
                             * to an InputEvent structure from which to
                             * create the new input event.  In this example,
                             * the pointer is NULL, meaning no new event
                             * should be introduced.
                             */
                            if (translate = (CxTranslate(NULL)))
                            {
                                AttachCxObj(filter, translate);

                                /* CxObjError() is a commodities.library
                                 * function that returns the internal
                                 * accumulated error code of a CxObject.
                                 */
                                if (! CxObjError(filter))
                                {
                                    ActivateCxObj(broker, 1L);
                                    while (ProcessMsg());
                                }
                            }
                        }
                    }
                    /* DeleteCxObjAll() is a commodities.library function
                     * that not only deletes the CxObject pointed to in
                     * its argument, but it deletes all of the CxObjects
                     * that are attached to it.
                     */
                    DeleteCxObjAll(broker);
                }
                DeleteMsgPort(broker_mp);
            }
```

```
        /* this amiga.lib function cleans up after ArgArrayInit() */
        ArgArrayDone();
        CloseLibrary(IconBase);
    }
    CloseLibrary(CxBase);
  }
}



LONG ProcessMsg(void)
{
    extern struct MsgPort *broker_mp;
    extern CxObj *broker;

    extern ULONG cxsigflag;

    CxMsg *msg;

    ULONG sigrcvd, msgid, msgtype;
    LONG returnvalue = 1L;

    sigrcvd = Wait(SIGBREAKF_CTRL_C | cxsigflag);

    while(msg = (CxMsg *)GetMsg(broker_mp))
    {
        msgid = CxMsgID(msg);
        msgtype = CxMsgType(msg);
        ReplyMsg((struct Message *)msg);

        switch(msgtype)
        {
            case CXM_IEVENT:
                printf("A CXM_EVENT, ");
                switch(msgid)
                {
                    case EVT_HOTKEY:
                    /* We got the message from the sender CxObject */
                        printf("You hit the HotKey.\n");
                        break;

                    default:
                        printf("unknown.\n");
                        break;
                }
                break;

            case CXM_COMMAND:
                printf("A command: ");
                switch(msgid)
                {
                    case CXCMD_DISABLE:
                        printf("CXCMD_DISABLE\n");
                        ActivateCxObj(broker, 0L);
                        break;

                    case CXCMD_ENABLE:
                        printf("CXCMD_ENABLE\n");
                        ActivateCxObj(broker, 1L);
                        break;

                    case CXCMD_KILL:
                        printf("CXCMD_KILL\n");
                        returnvalue = 0L;
                        break;

                    case CXCMD_UNIQUE:
                    /* Commodities Exchange can be told not
                     * only to refuse to launch a commodity with
                     * a name already in use but also can notify
                     * the already running commodity that it happened.
                     * It does this by sending a CXM_COMMAND with the
                     * ID set to CXMCMD_UNIQUE.  If the user tries
                     * to run a windowless commodity that is already
                     * running, the user wants the commodity to shut down.
                     */
                        printf("CXCMD_UNIQUE\n");
                        returnvalue = 0L;
                        break;
                    default:
                        printf("Unknown msgid\n");
                        break;
                }
                break;
            default:
                printf("Unknown msgtype\n");
                break;
        }
    }

    if (sigrcvd & SIGBREAKF_CTRL_C)
    {
        returnvalue = 0L;
        printf("CTRL C signal break\n");
    }

    return(returnvalue);
}
```